EXHIBIT 3

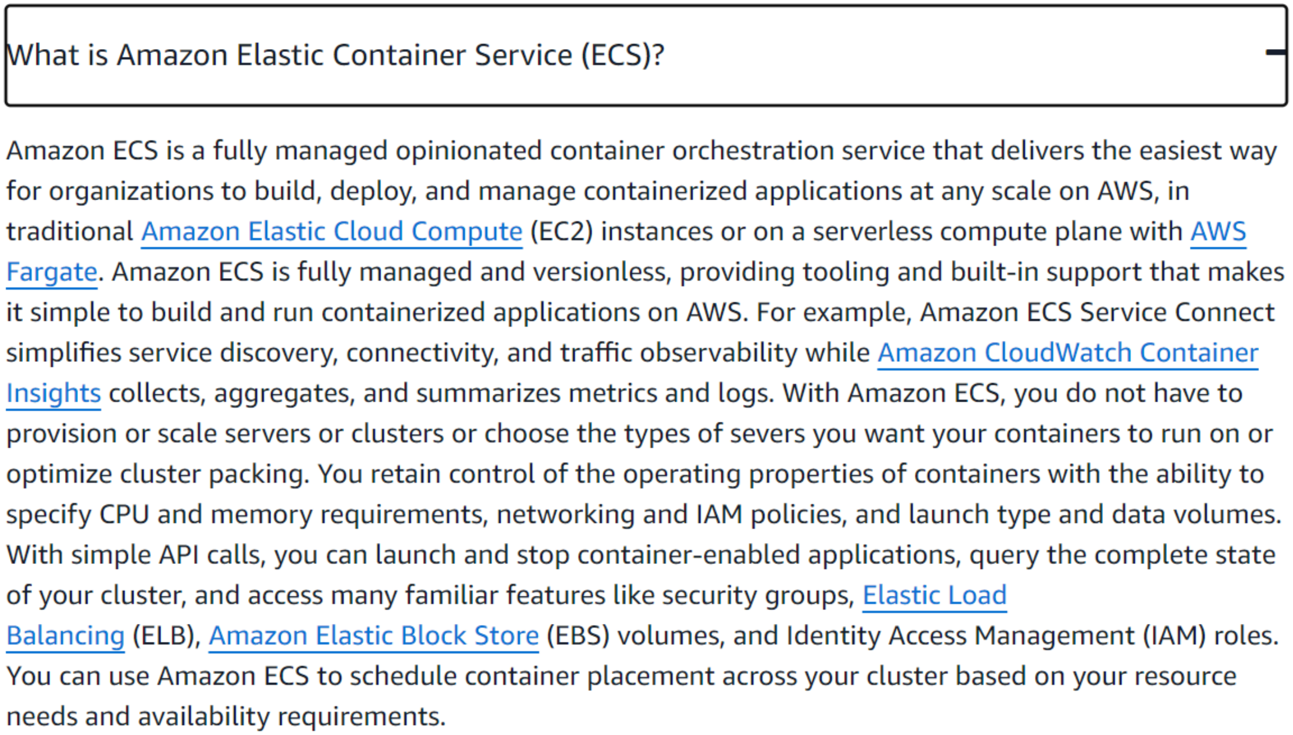
U.S. Patent No. 7,784,058 ("'058 Patent")

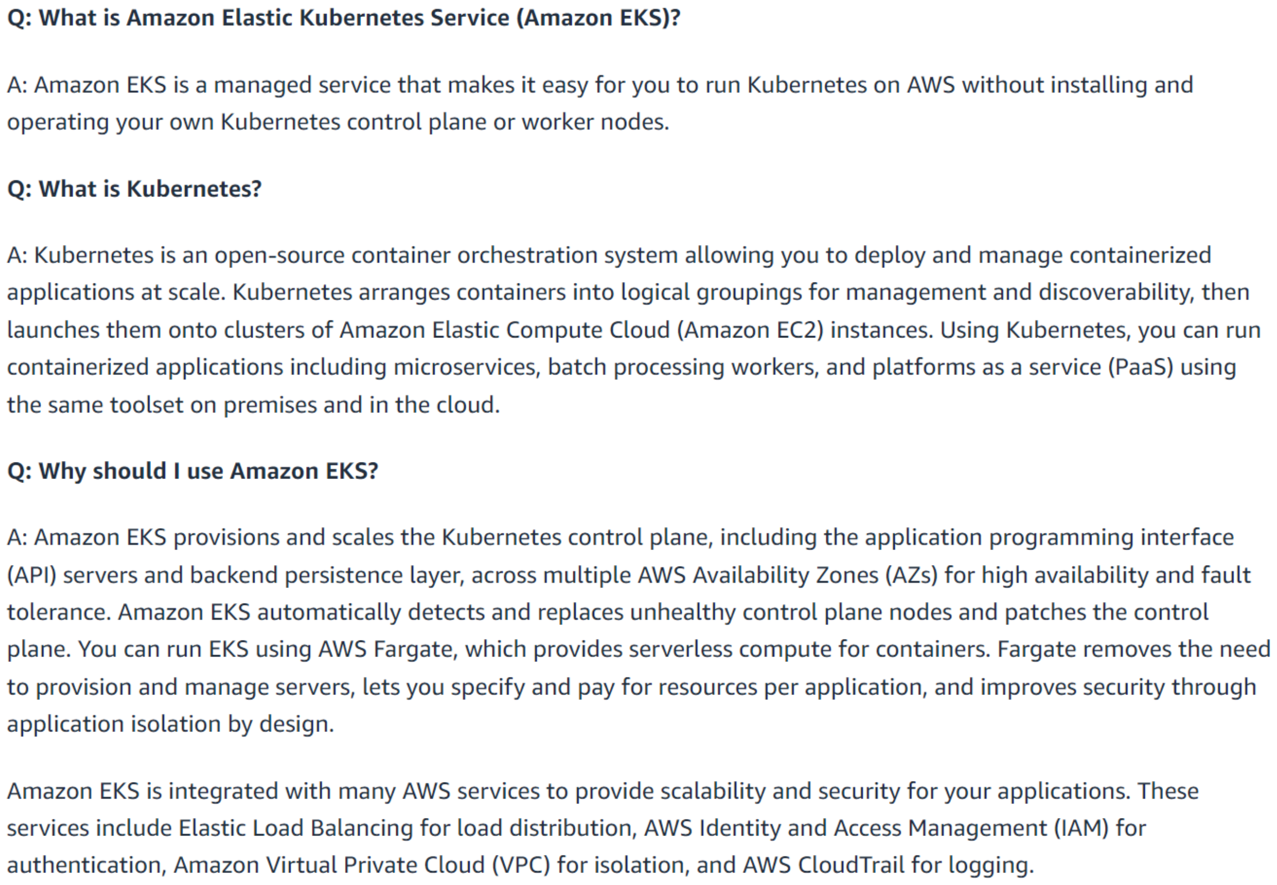
Accused Instrumentalities: Amazon products and services using user mode critical system elements as shared libraries, including without limitation Amazon AWS Elastic Container Service (ECS) (including ECS Anywhere), AWS Elastic Kubernetes Service (EKS) (including EKS Anywhere), AWS EC2 (including spot instances), AWS Elastic Container Registry (ECR), and AWS App2Container, and all versions and variations thereof since the issuance of the asserted patent.

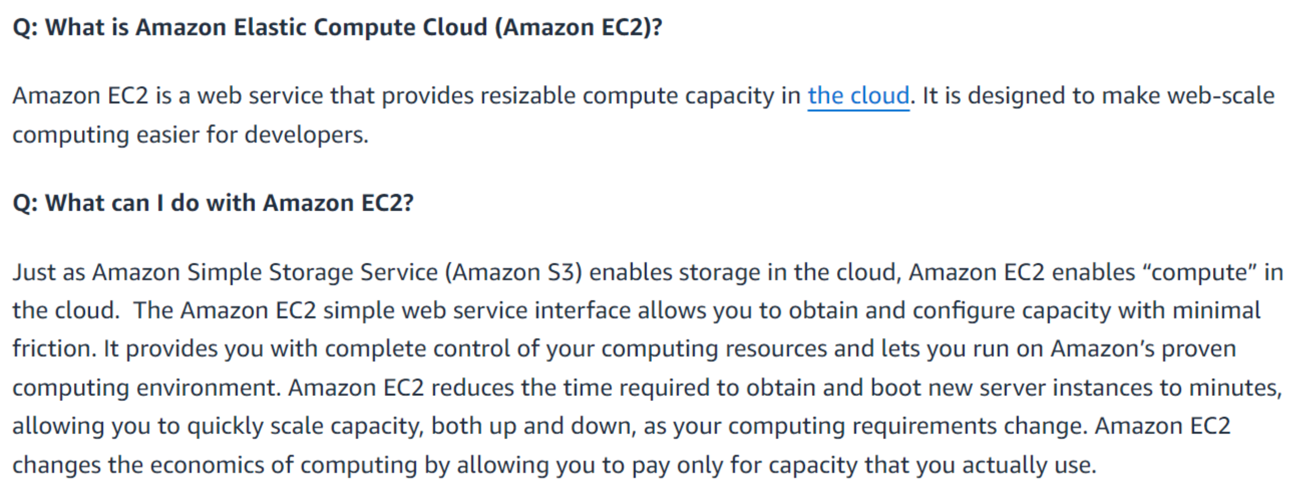
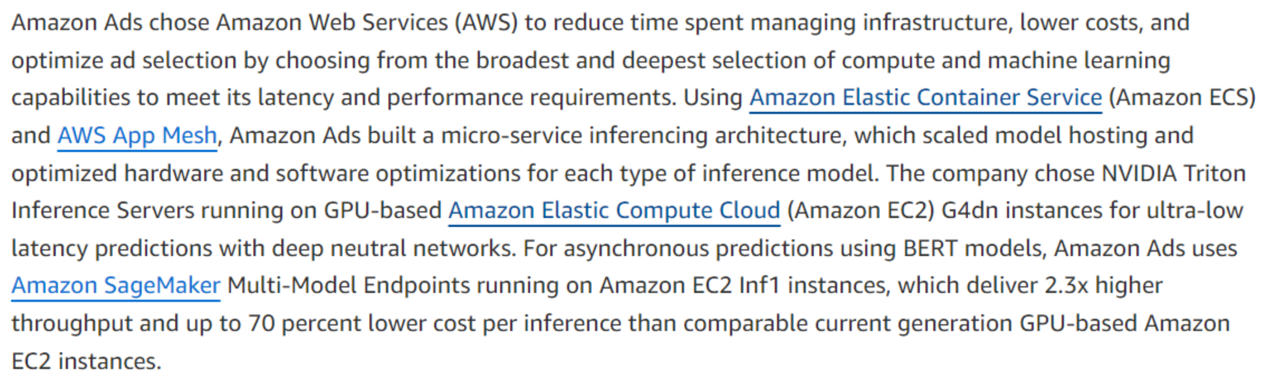
Each Accused Instrumentality, separately and in conjunction with each other as described below, infringes the claims in substantially the same way, and the evidence shown in this chart is similarly applicable to each Accused Instrumentality. Each claim limitation is literally infringed by each Accused Instrumentality. However, to the extent any claim limitation is not met literally, it is nonetheless met under the doctrine of equivalents because the differences between the claim limitation and each Accused Instrumentality would be insubstantial, and each Accused Instrumentality performs substantially the same function, in substantially the same way, to achieve the same result as the claimed invention. Notably, Defendant has not yet articulated which, if any, particular claim limitations it believes are not met by the Accused Instrumentalities.

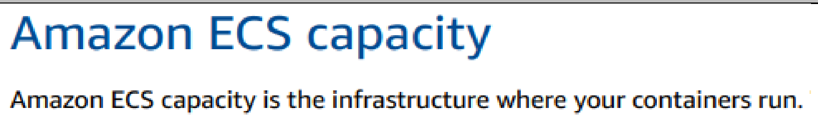
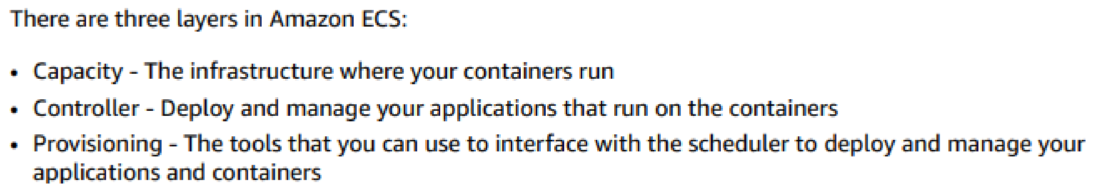
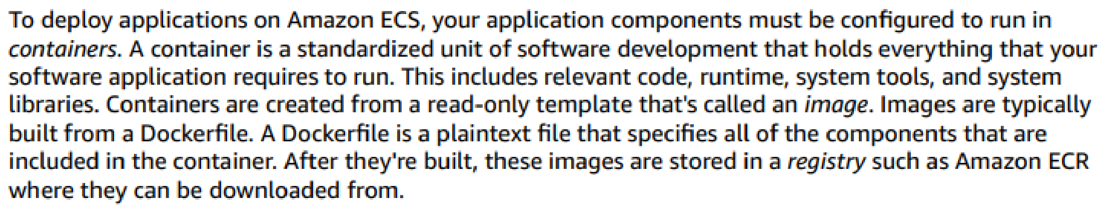
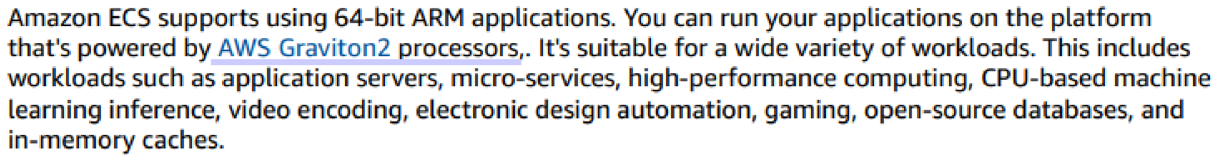
Claim 1

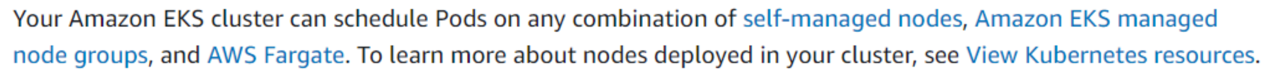
Claim 1	Accused Instrumentalities
[1pre] 1. A computing system for executing a plurality of software applications comprising:	To the extent the preamble is limiting, each Accused Instrumentality comprise or constitute a computing system for executing a plurality of software applications as claimed. *See* claim limitations below. *See also, e.g.*:

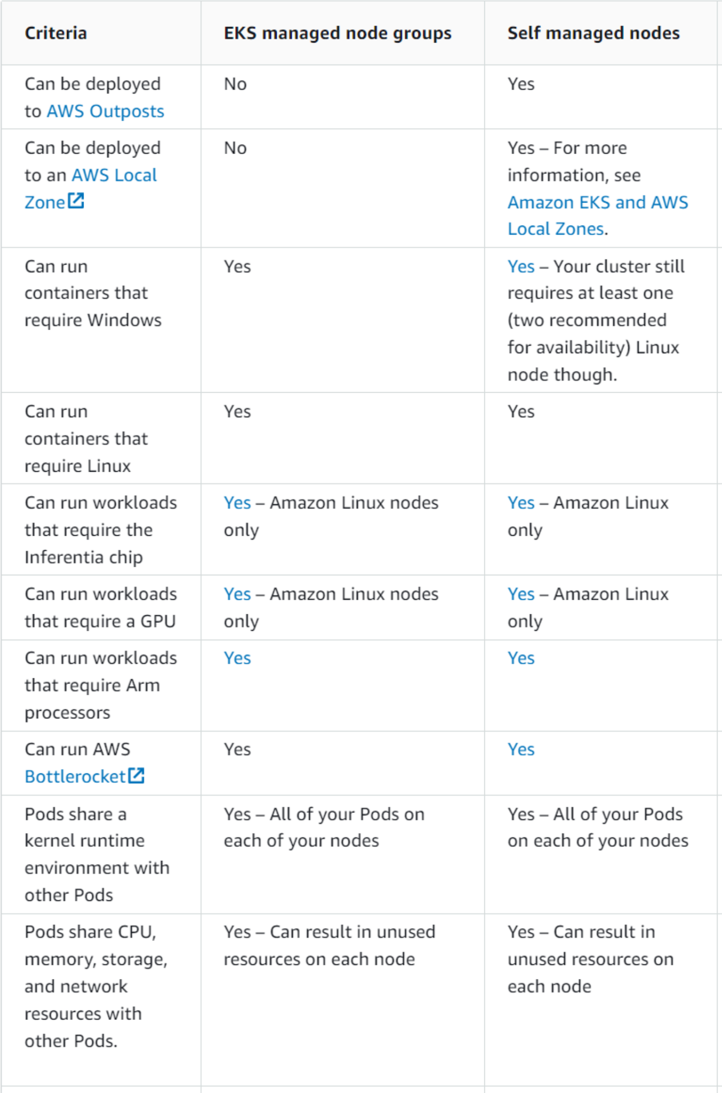
Claim 1	Accused Instrumentalities
	 What is Amazon Elastic Container Service (ECS)? Amazon ECS is a fully managed opinionated container orchestration service that delivers the easiest way for organizations to build, deploy, and manage containerized applications at any scale on AWS, in traditional Amazon Elastic Cloud Compute (EC2) instances or on a serverless compute plane with AWS Fargate. Amazon ECS is fully managed and versionless, providing tooling and built-in support that makes it simple to build and run containerized applications on AWS. For example, Amazon ECS Service Connect simplifies service discovery, connectivity, and traffic observability while Amazon CloudWatch Container Insights collects, aggregates, and summarizes metrics and logs. With Amazon ECS, you do not have to provision or scale servers or clusters or choose the types of severs you want your containers to run on or optimize cluster packing. You retain control of the operating properties of containers with the ability to specify CPU and memory requirements, networking and IAM policies, and launch type and data volumes. With simple API calls, you can launch and stop container-enabled applications, query the complete state of your cluster, and access many familiar features like security groups, Elastic Load Balancing (ELB), Amazon Elastic Block Store (EBS) volumes, and Identity Access Management (IAM) roles. You can use Amazon ECS to schedule container placement across your cluster based on your resource needs and availability requirements. https://aws.amazon.com/ecs/faqs/

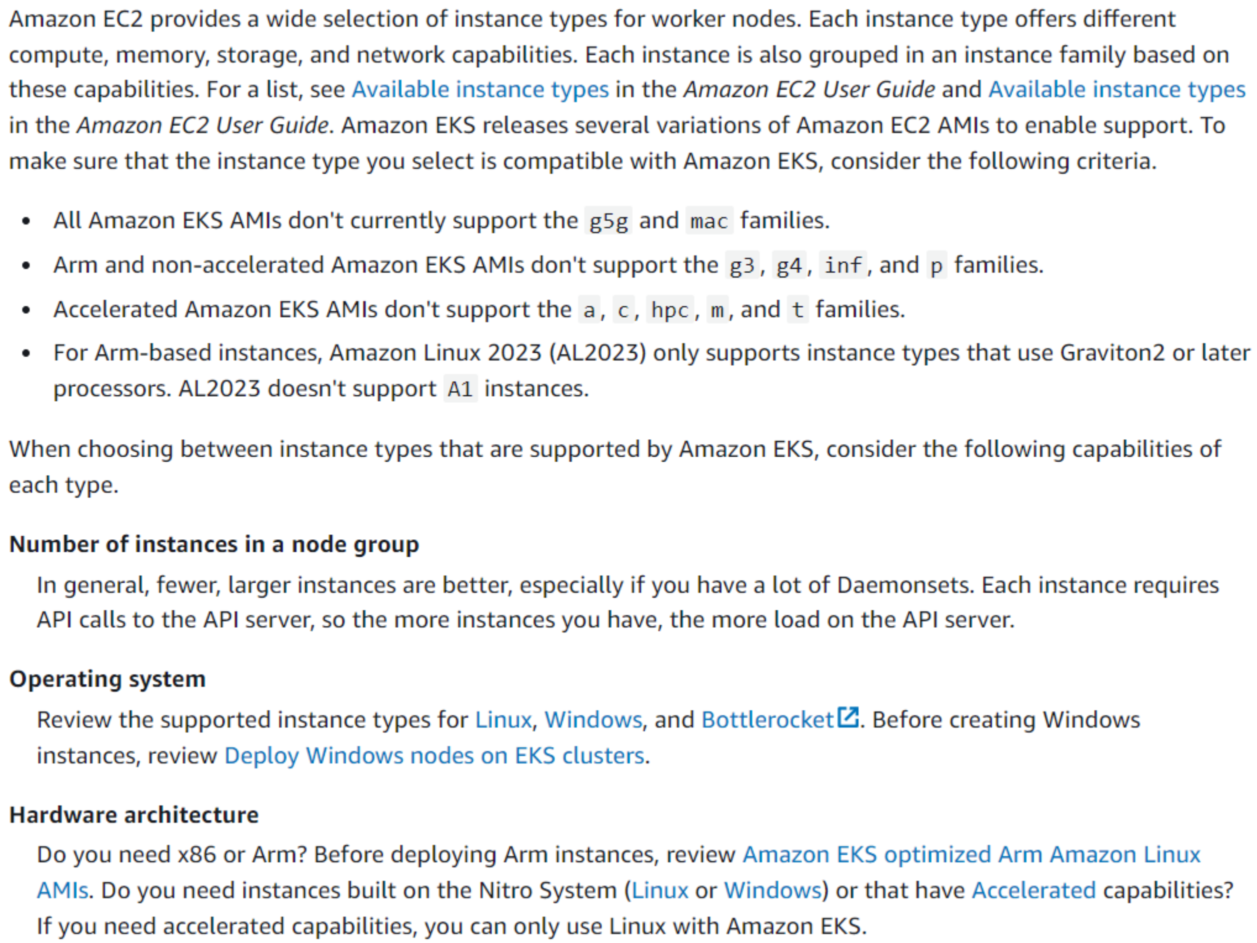
Claim 1	Accused Instrumentalities
	 **Q: What is Amazon Elastic Kubernetes Service (Amazon EKS)?** A: Amazon EKS is a managed service that makes it easy for you to run Kubernetes on AWS without installing and operating your own Kubernetes control plane or worker nodes. **Q: What is Kubernetes?** A: Kubernetes is an open-source container orchestration system allowing you to deploy and manage containerized applications at scale. Kubernetes arranges containers into logical groupings for management and discoverability, then launches them onto clusters of Amazon Elastic Compute Cloud (Amazon EC2) instances. Using Kubernetes, you can run containerized applications including microservices, batch processing workers, and platforms as a service (PaaS) using the same toolset on premises and in the cloud. **Q: Why should I use Amazon EKS?** A: Amazon EKS provisions and scales the Kubernetes control plane, including the application programming interface (API) servers and backend persistence layer, across multiple AWS Availability Zones (AZs) for high availability and fault tolerance. Amazon EKS automatically detects and replaces unhealthy control plane nodes and patches the control plane. You can run EKS using AWS Fargate, which provides serverless compute for containers. Fargate removes the need to provision and manage servers, lets you specify and pay for resources per application, and improves security through application isolation by design. Amazon EKS is integrated with many AWS services to provide scalability and security for your applications. These services include Elastic Load Balancing for load distribution, AWS Identity and Access Management (IAM) for authentication, Amazon Virtual Private Cloud (VPC) for isolation, and AWS CloudTrail for logging. https://aws.amazon.com/eks/faqs/

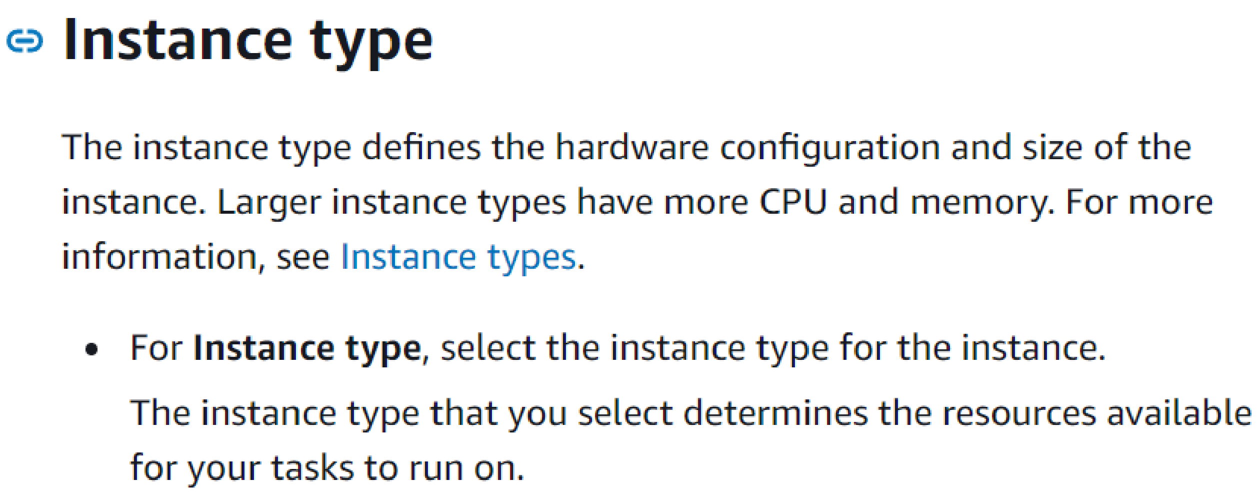
Claim 1	Accused Instrumentalities
	 **Q: What is Amazon Elastic Compute Cloud (Amazon EC2)?** Amazon EC2 is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers. **Q: What can I do with Amazon EC2?** Just as Amazon Simple Storage Service (Amazon S3) enables storage in the cloud, Amazon EC2 enables “compute” in the cloud. The Amazon EC2 simple web service interface allows you to obtain and configure capacity with minimal friction. It provides you with complete control of your computing resources and lets you run on Amazon’s proven computing environment. Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing you to quickly scale capacity, both up and down, as your computing requirements change. Amazon EC2 changes the economics of computing by allowing you to pay only for capacity that you actually use. https://aws.amazon.com/ec2/faqs/  Amazon Ads chose Amazon Web Services (AWS) to reduce time spent managing infrastructure, lower costs, and optimize ad selection by choosing from the broadest and deepest selection of compute and machine learning capabilities to meet its latency and performance requirements. Using Amazon Elastic Container Service (Amazon ECS) and AWS App Mesh, Amazon Ads built a micro-service inferencing architecture, which scaled model hosting and optimized hardware and software optimizations for each type of inference model. The company chose NVIDIA Triton Inference Servers running on GPU-based Amazon Elastic Compute Cloud (Amazon EC2) G4dn instances for ultra-low latency predictions with deep neutral networks. For asynchronous predictions using BERT models, Amazon Ads uses Amazon SageMaker Multi-Model Endpoints running on Amazon EC2 Inf1 instances, which deliver 2.3x higher throughput and up to 70 percent lower cost per inference than comparable current generation GPU-based Amazon EC2 instances. https://aws.amazon.com/solutions/case-studies/amazonads-kunliu-video-case-study/?did=cr_card&trk=cr_card

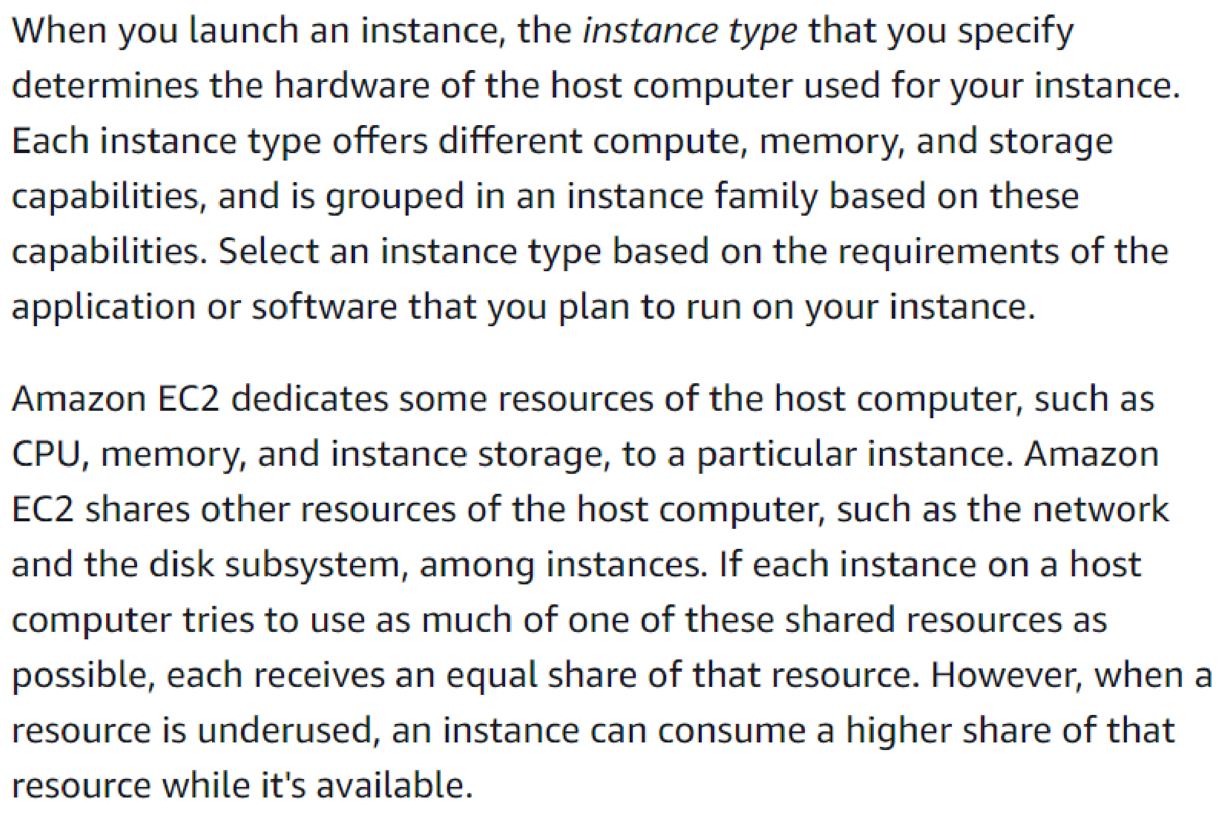
Claim 1	Accused Instrumentalities
	 Amazon ECS capacity Amazon ECS capacity is the infrastructure where your containers run. https://docs.aws.amazon.com/pdfs/AmazonECS/latest/developerguide/ecs-dg.pdf  There are three layers in Amazon ECS: • Capacity - The infrastructure where your containers run • Controller - Deploy and manage your applications that run on the containers • Provisioning - The tools that you can use to interface with the scheduler to deploy and manage your applications and containers https://docs.aws.amazon.com/pdfs/AmazonECS/latest/developerguide/ecs-dg.pdf  To deploy applications on Amazon ECS, your application components must be configured to run in *containers*. A container is a standardized unit of software development that holds everything that your software application requires to run. This includes relevant code, runtime, system tools, and system libraries. Containers are created from a read-only template that's called an *image*. Images are typically built from a Dockerfile. A Dockerfile is a plaintext file that specifies all of the components that are included in the container. After they're built, these images are stored in a *registry* such as Amazon ECR where they can be downloaded from. https://docs.aws.amazon.com/pdfs/AmazonECS/latest/developerguide/ecs-dg.pdf
[1a] a) a processor;	Each Accused Instrumentality comprises a processor. *See, e.g.*:  Amazon ECS supports using 64-bit ARM applications. You can run your applications on the platform that's powered by AWS Graviton2 processors,. It's suitable for a wide variety of workloads. This includes workloads such as application servers, micro-services, high-performance computing, CPU-based machine learning inference, video encoding, electronic design automation, gaming, open-source databases, and in-memory caches. https://docs.aws.amazon.com/pdfs/AmazonECS/latest/developerguide/ecs-dg.pdf (annotated)

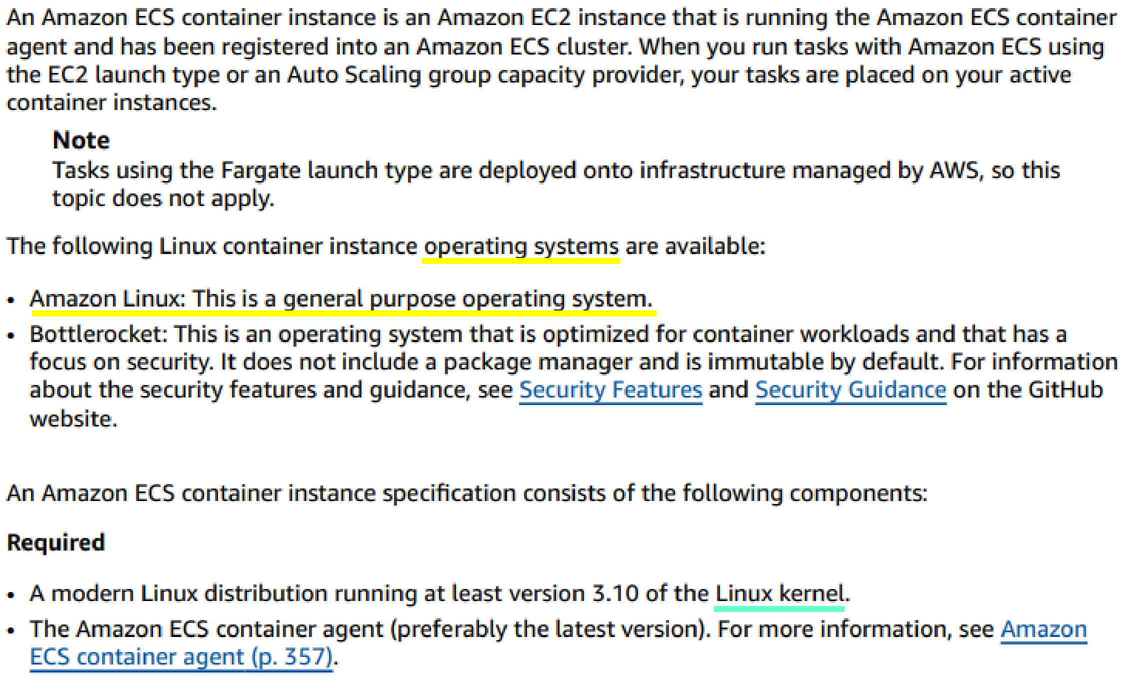
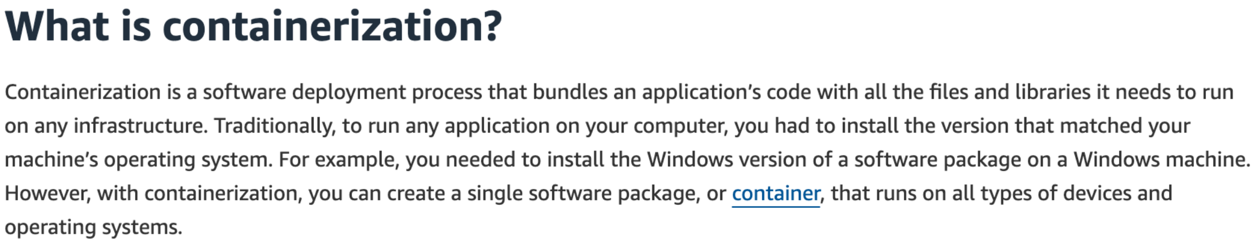
Claim 1	Accused Instrumentalities
	 Your Amazon EKS cluster can schedule Pods on any combination of self-managed nodes, Amazon EKS managed node groups, and AWS Fargate. To learn more about nodes deployed in your cluster, see View Kubernetes resources. https://docs.aws.amazon.com/eks/latest/userguide/eks-compute.html

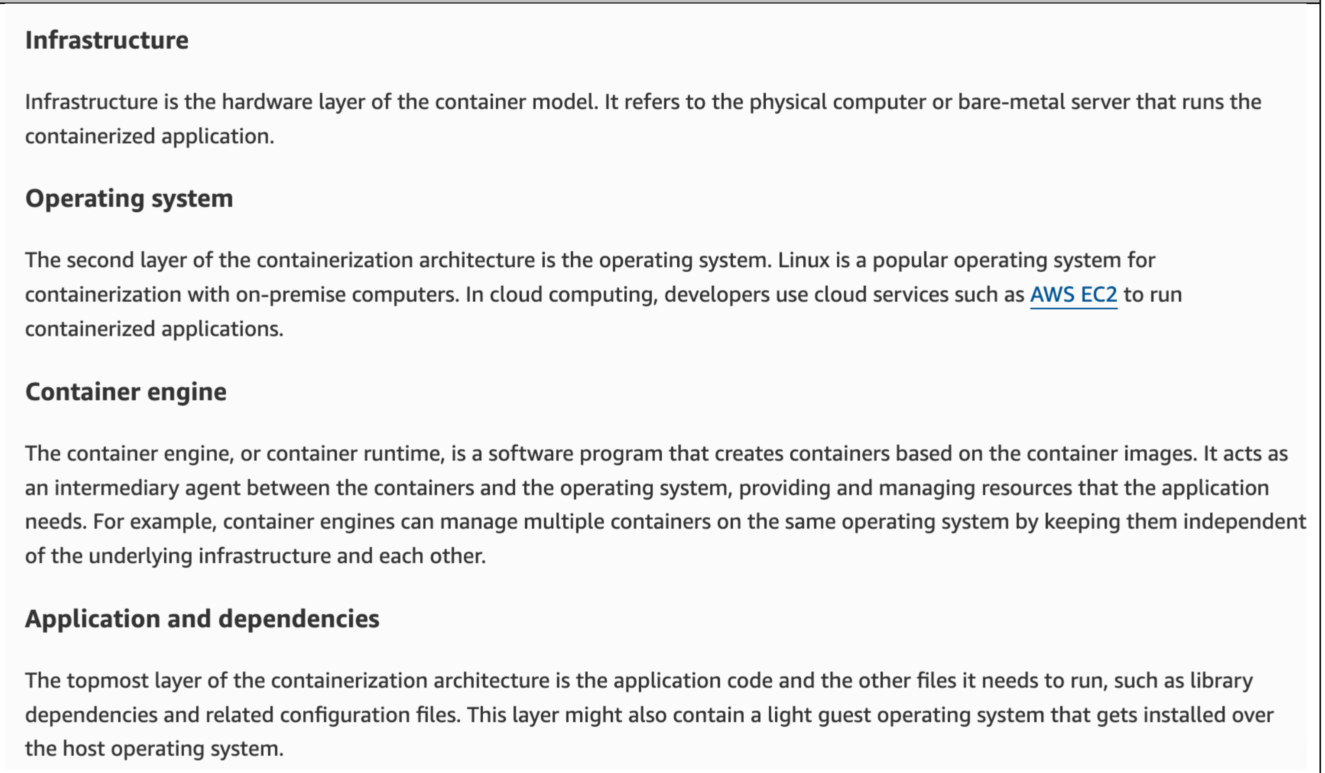
<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

<table>
<tr><th>Criteria</th><th>EKS managed node groups</th><th>Self managed nodes</th></tr>
<tr><td>Can be deployed to AWS Outposts</td><td>No</td><td>Yes</td></tr>
<tr><td>Can be deployed to an AWS Local Zone</td><td>No</td><td>Yes – For more information, see Amazon EKS and AWS Local Zones.</td></tr>
<tr><td>Can run containers that require Windows</td><td>Yes</td><td>Yes – Your cluster still requires at least one (two recommended for availability) Linux node though.</td></tr>
<tr><td>Can run containers that require Linux</td><td>Yes</td><td>Yes</td></tr>
<tr><td>Can run workloads that require the Inferentia chip</td><td>Yes – Amazon Linux nodes only</td><td>Yes – Amazon Linux only</td></tr>
<tr><td>Can run workloads that require a GPU</td><td>Yes – Amazon Linux nodes only</td><td>Yes – Amazon Linux only</td></tr>
<tr><td>Can run workloads that require Arm processors</td><td>Yes</td><td>Yes</td></tr>
<tr><td>Can run AWS Bottlerocket</td><td>Yes</td><td>Yes</td></tr>
<tr><td>Pods share a kernel runtime environment with other Pods</td><td>Yes – All of your Pods on each of your nodes</td><td>Yes – All of your Pods on each of your nodes</td></tr>
<tr><td>Pods share CPU, memory, storage, and network resources with other Pods.</td><td>Yes – Can result in unused resources on each node</td><td>Yes – Can result in unused resources on each node</td></tr>
</table>
https://docs.aws.amazon.com/eks/latest/userguide/eks-compute.html
</td></tr>
</table>

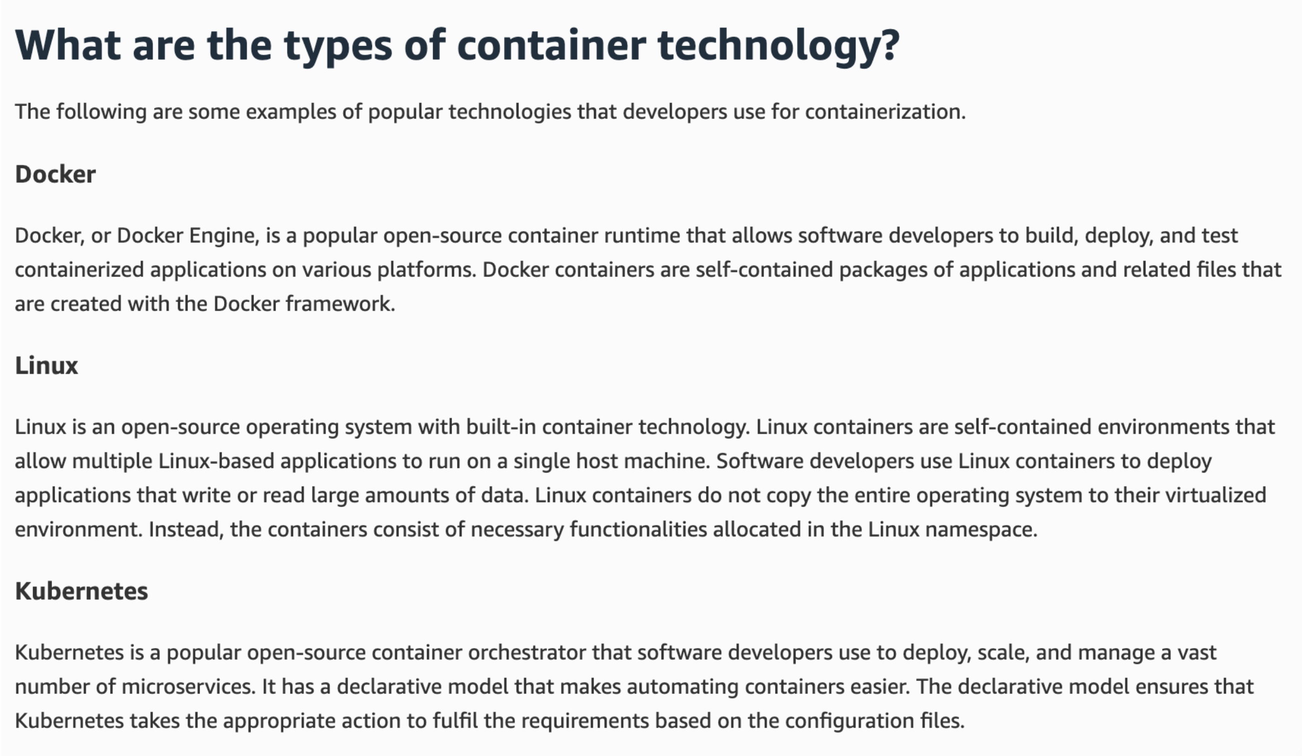
Claim 1	Accused Instrumentalities
	 Amazon EC2 provides a wide selection of instance types for worker nodes. Each instance type offers different compute, memory, storage, and network capabilities. Each instance is also grouped in an instance family based on these capabilities. For a list, see Available instance types in the *Amazon EC2 User Guide* and Available instance types in the *Amazon EC2 User Guide*. Amazon EKS releases several variations of Amazon EC2 AMIs to enable support. To make sure that the instance type you select is compatible with Amazon EKS, consider the following criteria. - All Amazon EKS AMIs don't currently support the `g5g` and `mac` families. - Arm and non-accelerated Amazon EKS AMIs don't support the `g3`, `g4`, `inf`, and `p` families. - Accelerated Amazon EKS AMIs don't support the `a`, `c`, `hpc`, `m`, and `t` families. - For Arm-based instances, Amazon Linux 2023 (AL2023) only supports instance types that use Graviton2 or later processors. AL2023 doesn't support `A1` instances. When choosing between instance types that are supported by Amazon EKS, consider the following capabilities of each type. **Number of instances in a node group** In general, fewer, larger instances are better, especially if you have a lot of Daemonsets. Each instance requires API calls to the API server, so the more instances you have, the more load on the API server. **Operating system** Review the supported instance types for Linux, Windows, and Bottlerocket. Before creating Windows instances, review Deploy Windows nodes on EKS clusters. **Hardware architecture** Do you need x86 or Arm? Before deploying Arm instances, review Amazon EKS optimized Arm Amazon Linux AMIs. Do you need instances built on the Nitro System (Linux or Windows) or that have Accelerated capabilities? If you need accelerated capabilities, you can only use Linux with Amazon EKS. https://docs.aws.amazon.com/eks/latest/userguide/choosing-instance-type.html

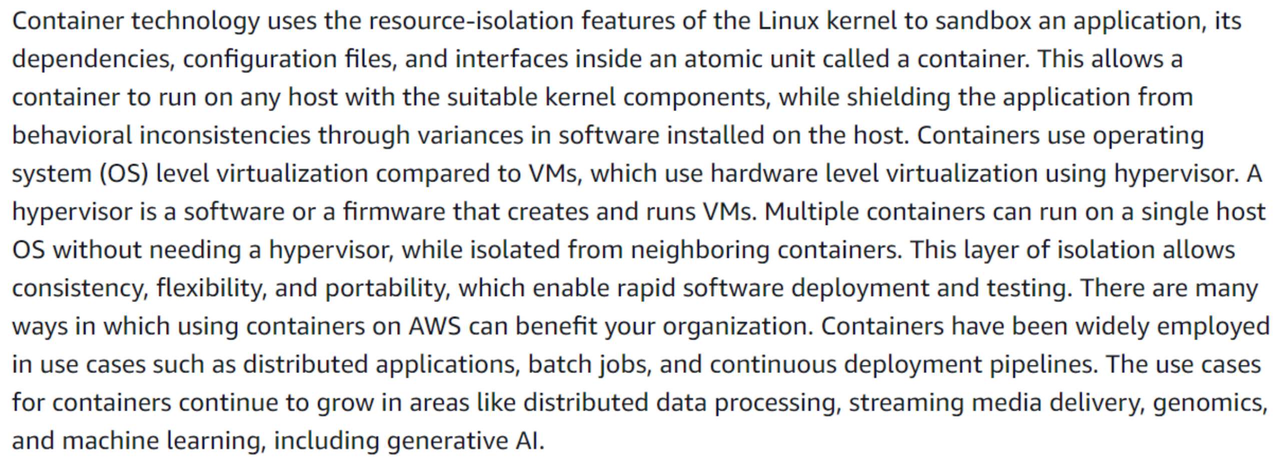
Claim 1	Accused Instrumentalities
	 **Instance type** The instance type defines the hardware configuration and size of the instance. Larger instance types have more CPU and memory. For more information, see Instance types. • For **Instance type**, select the instance type for the instance. The instance type that you select determines the resources available for your tasks to run on. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/launch_container_instance.html

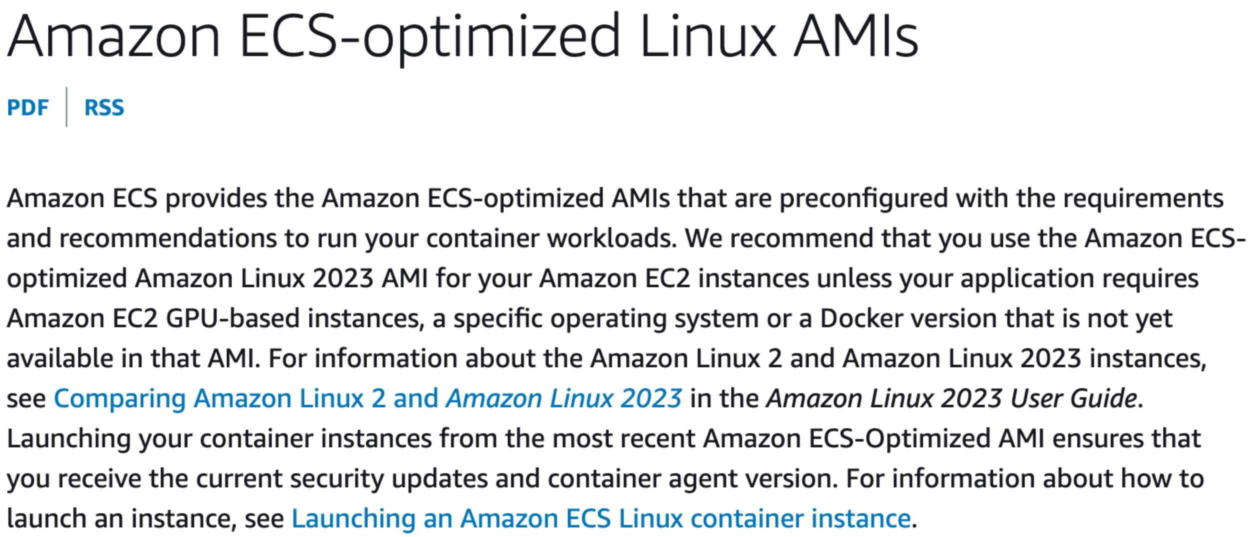
Claim 1	Accused Instrumentalities
	 When you launch an instance, the *instance type* that you specify determines the hardware of the host computer used for your instance. Each instance type offers different compute, memory, and storage capabilities, and is grouped in an instance family based on these capabilities. Select an instance type based on the requirements of the application or software that you plan to run on your instance. Amazon EC2 dedicates some resources of the host computer, such as CPU, memory, and instance storage, to a particular instance. Amazon EC2 shares other resources of the host computer, such as the network and the disk subsystem, among instances. If each instance on a host computer tries to use as much of one of these shared resources as possible, each receives an equal share of that resource. However, when a resource is underused, an instance can consume a higher share of that resource while it's available. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html
[1b] b) an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor; and,	Each Accused Instrumentality comprises an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor. *See, e.g.*:

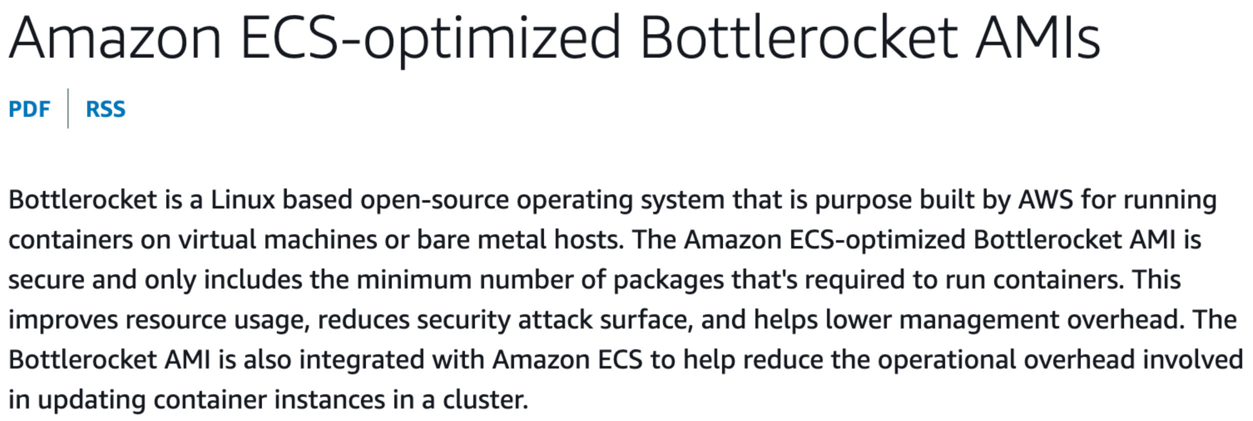
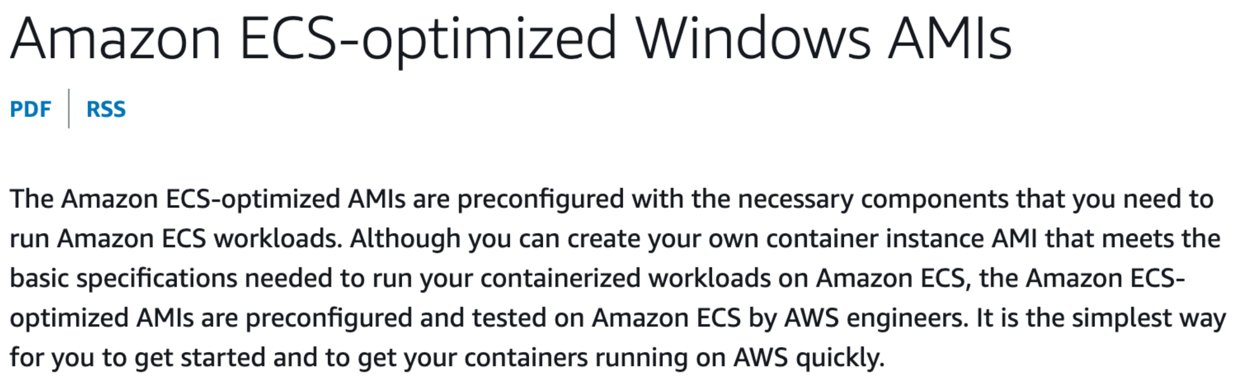
Claim 1	Accused Instrumentalities
	 An Amazon ECS container instance is an Amazon EC2 instance that is running the Amazon ECS container agent and has been registered into an Amazon ECS cluster. When you run tasks with Amazon ECS using the EC2 launch type or an Auto Scaling group capacity provider, your tasks are placed on your active container instances. **Note** Tasks using the Fargate launch type are deployed onto infrastructure managed by AWS, so this topic does not apply. The following Linux container instance operating systems are available: - Amazon Linux: This is a general purpose operating system. - Bottlerocket: This is an operating system that is optimized for container workloads and that has a focus on security. It does not include a package manager and is immutable by default. For information about the security features and guidance, see Security Features and Security Guidance on the GitHub website. An Amazon ECS container instance specification consists of the following components: **Required** - A modern Linux distribution running at least version 3.10 of the Linux kernel. - The Amazon ECS container agent (preferably the latest version). For more information, see Amazon ECS container agent (p. 357). https://docs.aws.amazon.com/pdfs/AmazonECS/latest/developerguide/ecs-dg.pdf (annotated)  **What is containerization?** Containerization is a software deployment process that bundles an application's code with all the files and libraries it needs to run on any infrastructure. Traditionally, to run any application on your computer, you had to install the version that matched your machine's operating system. For example, you needed to install the Windows version of a software package on a Windows machine. However, with containerization, you can create a single software package, or container, that runs on all types of devices and operating systems. https://aws.amazon.com/what-is/containerization/

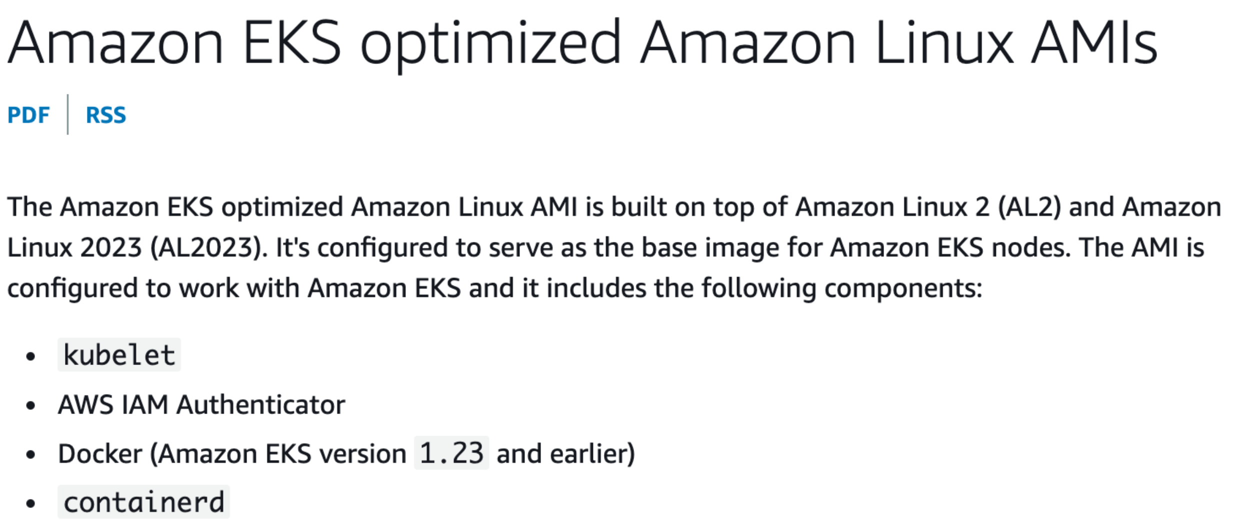
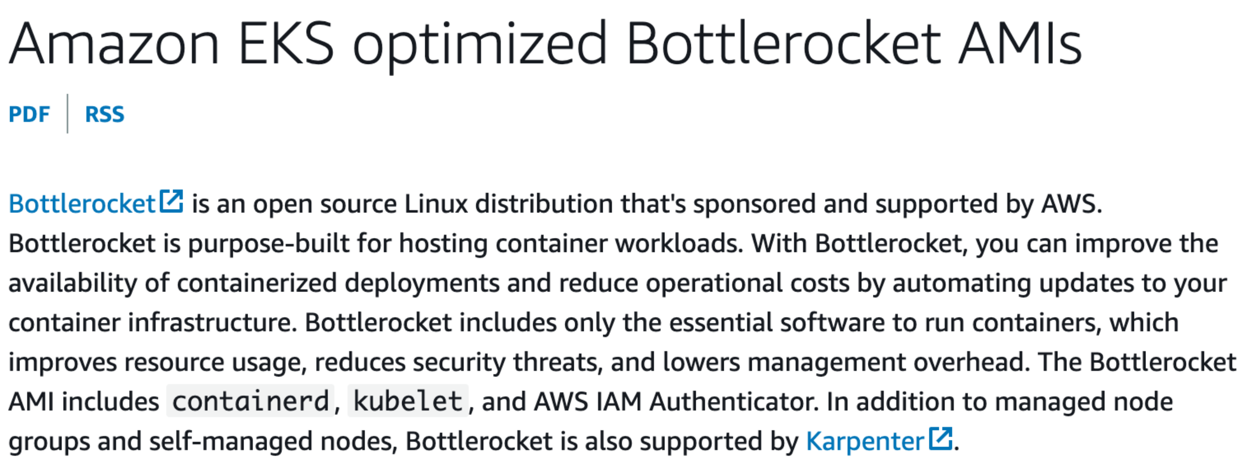
Claim 1	Accused Instrumentalities
	 **Infrastructure** Infrastructure is the hardware layer of the container model. It refers to the physical computer or bare-metal server that runs the containerized application. **Operating system** The second layer of the containerization architecture is the operating system. Linux is a popular operating system for containerization with on-premise computers. In cloud computing, developers use cloud services such as AWS EC2 to run containerized applications. **Container engine** The container engine, or container runtime, is a software program that creates containers based on the container images. It acts as an intermediary agent between the containers and the operating system, providing and managing resources that the application needs. For example, container engines can manage multiple containers on the same operating system by keeping them independent of the underlying infrastructure and each other. **Application and dependencies** The topmost layer of the containerization architecture is the application code and the other files it needs to run, such as library dependencies and related configuration files. This layer might also contain a light guest operating system that gets installed over the host operating system. https://aws.amazon.com/what-is/containerization/

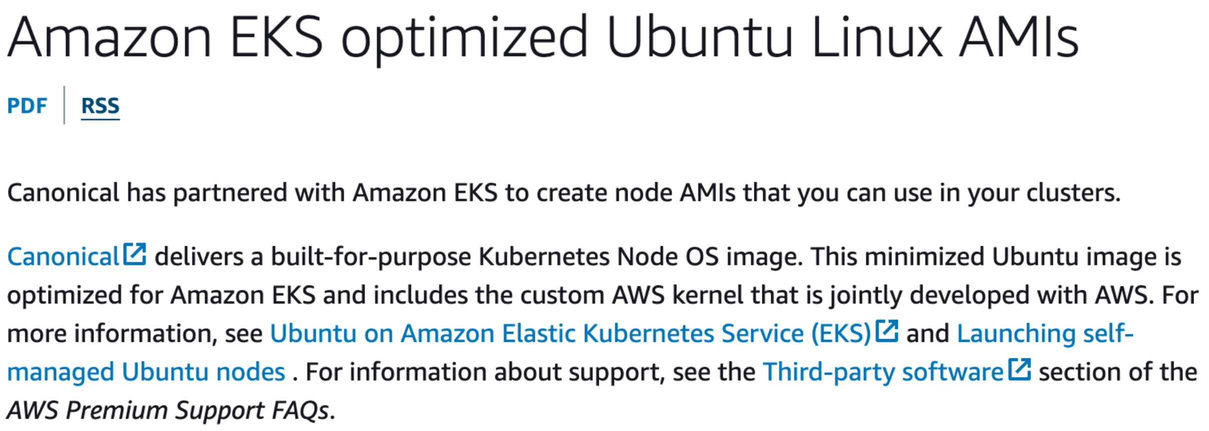
Claim 1	Accused Instrumentalities
	 **What are the types of container technology?** The following are some examples of popular technologies that developers use for containerization. **Docker** Docker, or Docker Engine, is a popular open-source container runtime that allows software developers to build, deploy, and test containerized applications on various platforms. Docker containers are self-contained packages of applications and related files that are created with the Docker framework. **Linux** Linux is an open-source operating system with built-in container technology. Linux containers are self-contained environments that allow multiple Linux-based applications to run on a single host machine. Software developers use Linux containers to deploy applications that write or read large amounts of data. Linux containers do not copy the entire operating system to their virtualized environment. Instead, the containers consist of necessary functionalities allocated in the Linux namespace. **Kubernetes** Kubernetes is a popular open-source container orchestrator that software developers use to deploy, scale, and manage a vast number of microservices. It has a declarative model that makes automating containers easier. The declarative model ensures that Kubernetes takes the appropriate action to fulfil the requirements based on the configuration files. https://aws.amazon.com/what-is/containerization/

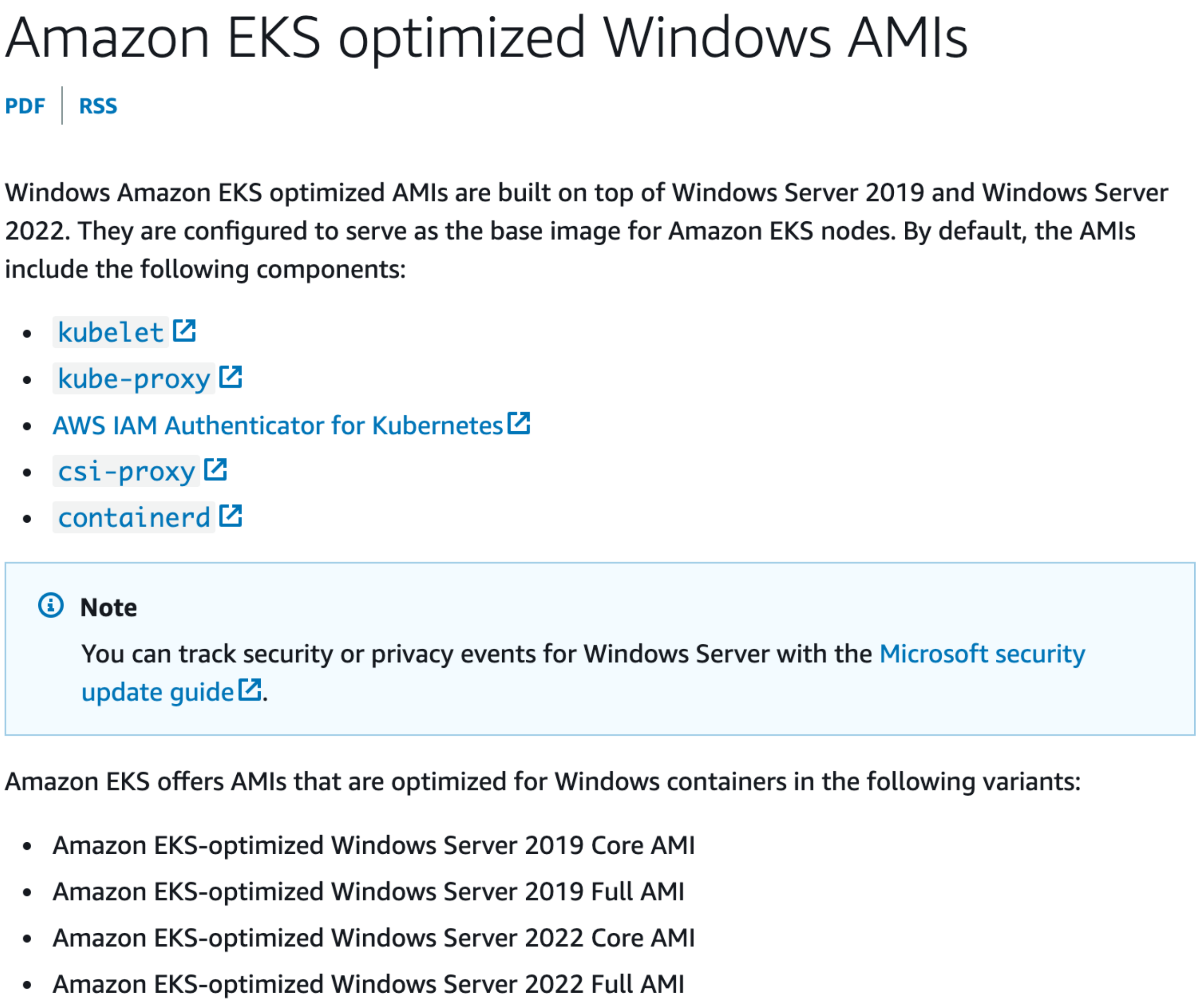
Claim 1	Accused Instrumentalities
	 Container technology uses the resource-isolation features of the Linux kernel to sandbox an application, its dependencies, configuration files, and interfaces inside an atomic unit called a container. This allows a container to run on any host with the suitable kernel components, while shielding the application from behavioral inconsistencies through variances in software installed on the host. Containers use operating system (OS) level virtualization compared to VMs, which use hardware level virtualization using hypervisor. A hypervisor is a software or a firmware that creates and runs VMs. Multiple containers can run on a single host OS without needing a hypervisor, while isolated from neighboring containers. This layer of isolation allows consistency, flexibility, and portability, which enable rapid software deployment and testing. There are many ways in which using containers on AWS can benefit your organization. Containers have been widely employed in use cases such as distributed applications, batch jobs, and continuous deployment pipelines. The use cases for containers continue to grow in areas like distributed data processing, streaming media delivery, genomics, and machine learning, including generative AI. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/containers-on-aws.html

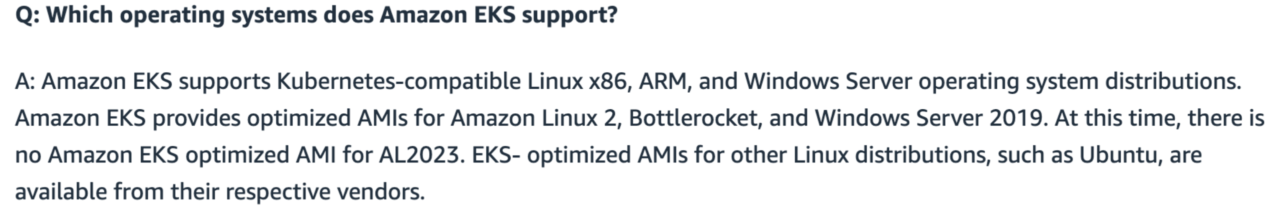
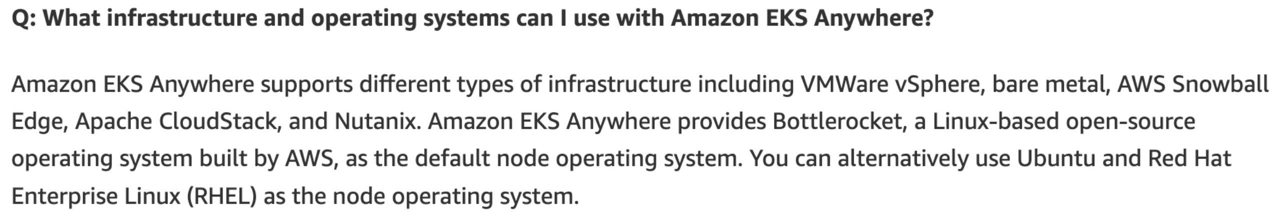
Claim 1	Accused Instrumentalities
	 Amazon ECS-optimized Linux AMIs PDF RSS Amazon ECS provides the Amazon ECS-optimized AMIs that are preconfigured with the requirements and recommendations to run your container workloads. We recommend that you use the Amazon ECS-optimized Amazon Linux 2023 AMI for your Amazon EC2 instances unless your application requires Amazon EC2 GPU-based instances, a specific operating system or a Docker version that is not yet available in that AMI. For information about the Amazon Linux 2 and Amazon Linux 2023 instances, see Comparing Amazon Linux 2 and *Amazon Linux 2023* in the *Amazon Linux 2023 User Guide*. Launching your container instances from the most recent Amazon ECS-Optimized AMI ensures that you receive the current security updates and container agent version. For information about how to launch an instance, see Launching an Amazon ECS Linux container instance. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-optimized_AMI.html; *see also* list of differences between Amazon Linux 2 and Amazon Linux 2023 at https://docs.aws.amazon.com/linux/al2023/ug/compare-with-al2.html

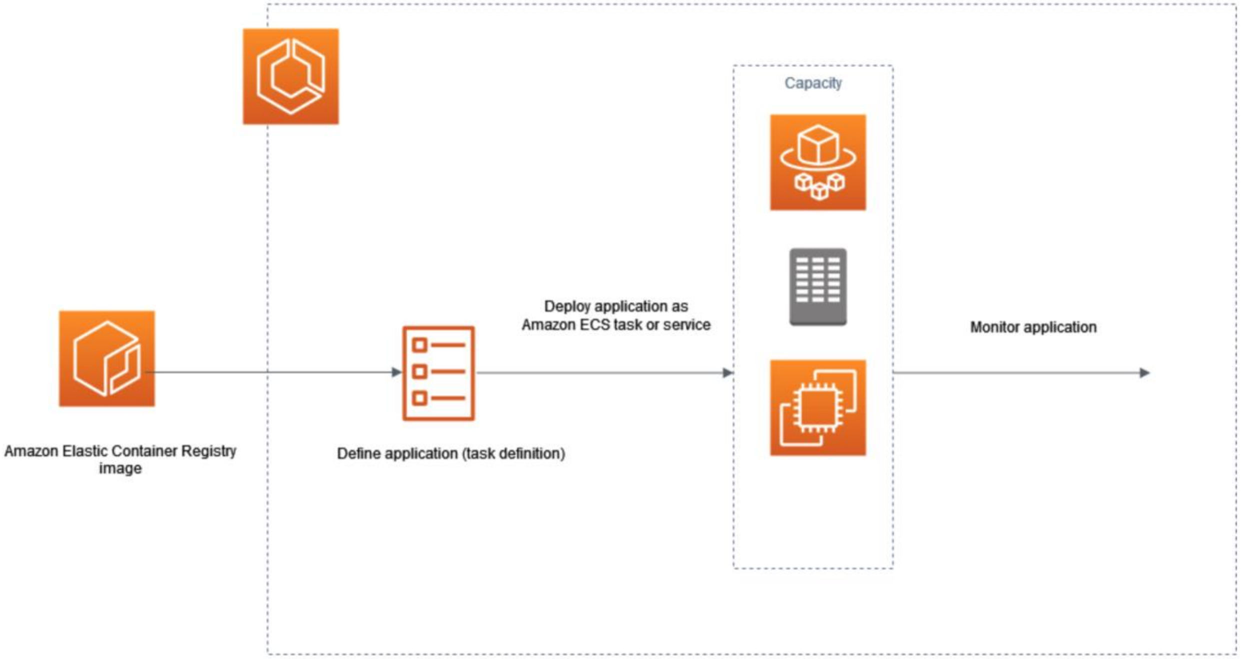
Claim 1	Accused Instrumentalities
	 Amazon ECS-optimized Bottlerocket AMIs PDF | RSS Bottlerocket is a Linux based open-source operating system that is purpose built by AWS for running containers on virtual machines or bare metal hosts. The Amazon ECS-optimized Bottlerocket AMI is secure and only includes the minimum number of packages that's required to run containers. This improves resource usage, reduces security attack surface, and helps lower management overhead. The Bottlerocket AMI is also integrated with Amazon ECS to help reduce the operational overhead involved in updating container instances in a cluster. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-bottlerocket.html  Amazon ECS-optimized Windows AMIs PDF | RSS The Amazon ECS-optimized AMIs are preconfigured with the necessary components that you need to run Amazon ECS workloads. Although you can create your own container instance AMI that meets the basic specifications needed to run your containerized workloads on Amazon ECS, the Amazon ECS-optimized AMIs are preconfigured and tested on Amazon ECS by AWS engineers. It is the simplest way for you to get started and to get your containers running on AWS quickly. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-optimized_windows_AMI.html

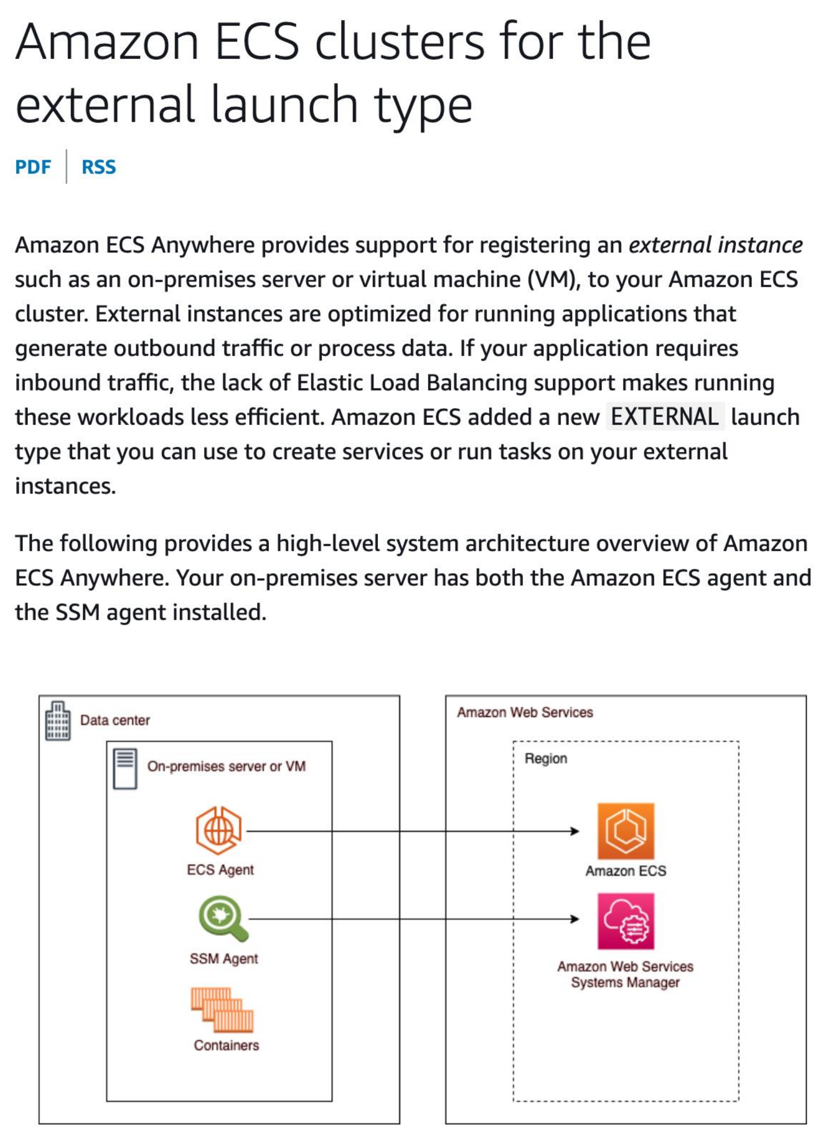
Claim 1	Accused Instrumentalities
	 Amazon EKS optimized Amazon Linux AMIs PDF RSS The Amazon EKS optimized Amazon Linux AMI is built on top of Amazon Linux 2 (AL2) and Amazon Linux 2023 (AL2023). It's configured to serve as the base image for Amazon EKS nodes. The AMI is configured to work with Amazon EKS and it includes the following components: • `kubelet` • AWS IAM Authenticator • Docker (Amazon EKS version `1.23` and earlier) • `containerd` https://docs.aws.amazon.com/eks/latest/userguide/eks-optimized-ami.html  Amazon EKS optimized Bottlerocket AMIs PDF RSS Bottlerocket is an open source Linux distribution that's sponsored and supported by AWS. Bottlerocket is purpose-built for hosting container workloads. With Bottlerocket, you can improve the availability of containerized deployments and reduce operational costs by automating updates to your container infrastructure. Bottlerocket includes only the essential software to run containers, which improves resource usage, reduces security threats, and lowers management overhead. The Bottlerocket AMI includes `containerd`, `kubelet`, and AWS IAM Authenticator. In addition to managed node groups and self-managed nodes, Bottlerocket is also supported by Karpenter. https://docs.aws.amazon.com/eks/latest/userguide/eks-optimized-ami-bottlerocket.html

Claim 1	Accused Instrumentalities
	 Amazon EKS optimized Ubuntu Linux AMIs PDF RSS Canonical has partnered with Amazon EKS to create node AMIs that you can use in your clusters. Canonical delivers a built-for-purpose Kubernetes Node OS image. This minimized Ubuntu image is optimized for Amazon EKS and includes the custom AWS kernel that is jointly developed with AWS. For more information, see Ubuntu on Amazon Elastic Kubernetes Service (EKS) and Launching self-managed Ubuntu nodes . For information about support, see the Third-party software section of the *AWS Premium Support FAQs*. https://docs.aws.amazon.com/eks/latest/userguide/eks-partner-amis.html

Claim 1	Accused Instrumentalities
	 Amazon EKS optimized Windows AMIs PDF | RSS Windows Amazon EKS optimized AMIs are built on top of Windows Server 2019 and Windows Server 2022. They are configured to serve as the base image for Amazon EKS nodes. By default, the AMIs include the following components: • kubelet • kube-proxy • AWS IAM Authenticator for Kubernetes • csi-proxy • containerd Note You can track security or privacy events for Windows Server with the Microsoft security update guide. Amazon EKS offers AMIs that are optimized for Windows containers in the following variants: • Amazon EKS-optimized Windows Server 2019 Core AMI • Amazon EKS-optimized Windows Server 2019 Full AMI • Amazon EKS-optimized Windows Server 2022 Core AMI • Amazon EKS-optimized Windows Server 2022 Full AMI https://docs.aws.amazon.com/eks/latest/userguide/eks-optimized-windows-ami.html

Claim 1	Accused Instrumentalities
	 **Q: Which operating systems does Amazon EKS support?** A: Amazon EKS supports Kubernetes-compatible Linux x86, ARM, and Windows Server operating system distributions. Amazon EKS provides optimized AMIs for Amazon Linux 2, Bottlerocket, and Windows Server 2019. At this time, there is no Amazon EKS optimized AMI for AL2023. EKS- optimized AMIs for other Linux distributions, such as Ubuntu, are available from their respective vendors. https://aws.amazon.com/eks/faqs/  **Q: What infrastructure and operating systems can I use with Amazon EKS Anywhere?** Amazon EKS Anywhere supports different types of infrastructure including VMWare vSphere, bare metal, AWS Snowball Edge, Apache CloudStack, and Nutanix. Amazon EKS Anywhere provides Bottlerocket, a Linux-based open-source operating system built by AWS, as the default node operating system. You can alternatively use Ubuntu and Red Hat Enterprise Linux (RHEL) as the node operating system. https://aws.amazon.com/eks/eks-anywhere/faqs/

Claim 1	Accused Instrumentalities
	Amazon ECS Application Lifecycle  Capacity Deploy application as Amazon ECS task or service Monitor application Amazon Elastic Container Registry image Define application (task definition) https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html



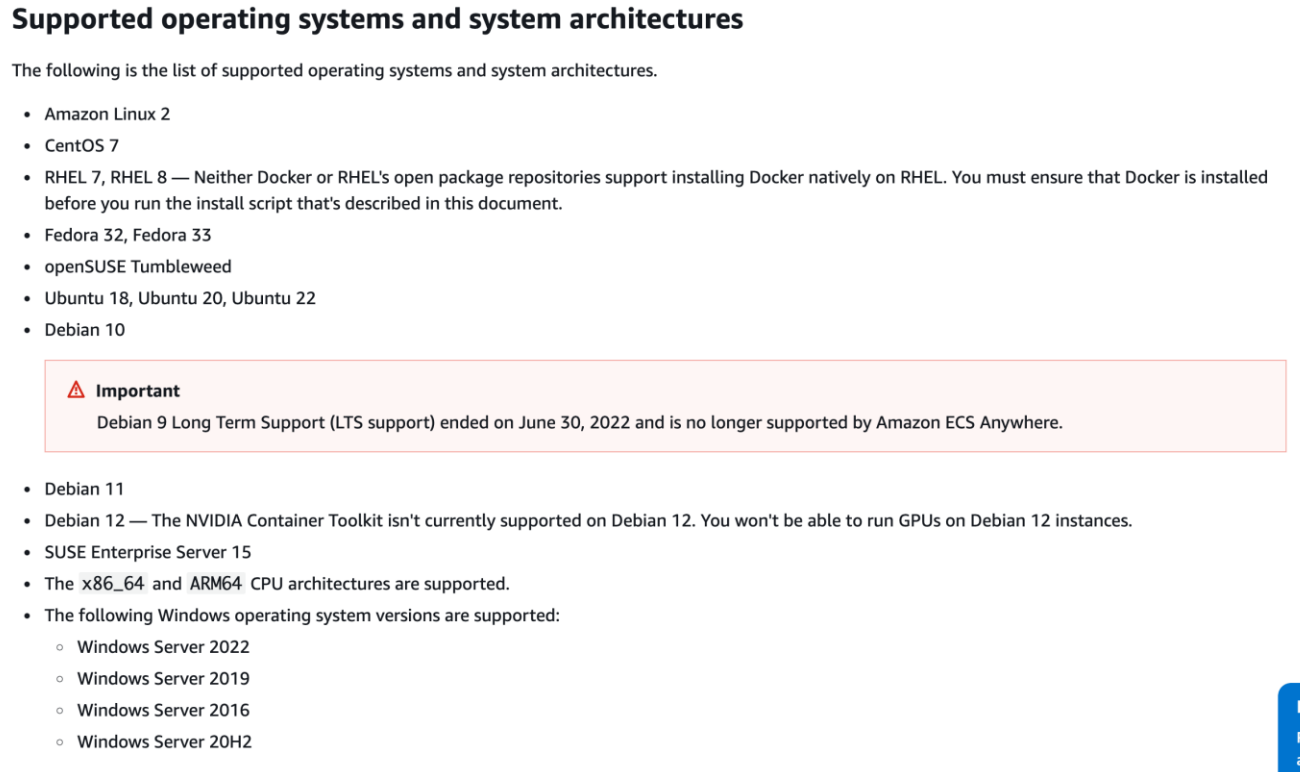
Amazon ECS clusters for the external launch type

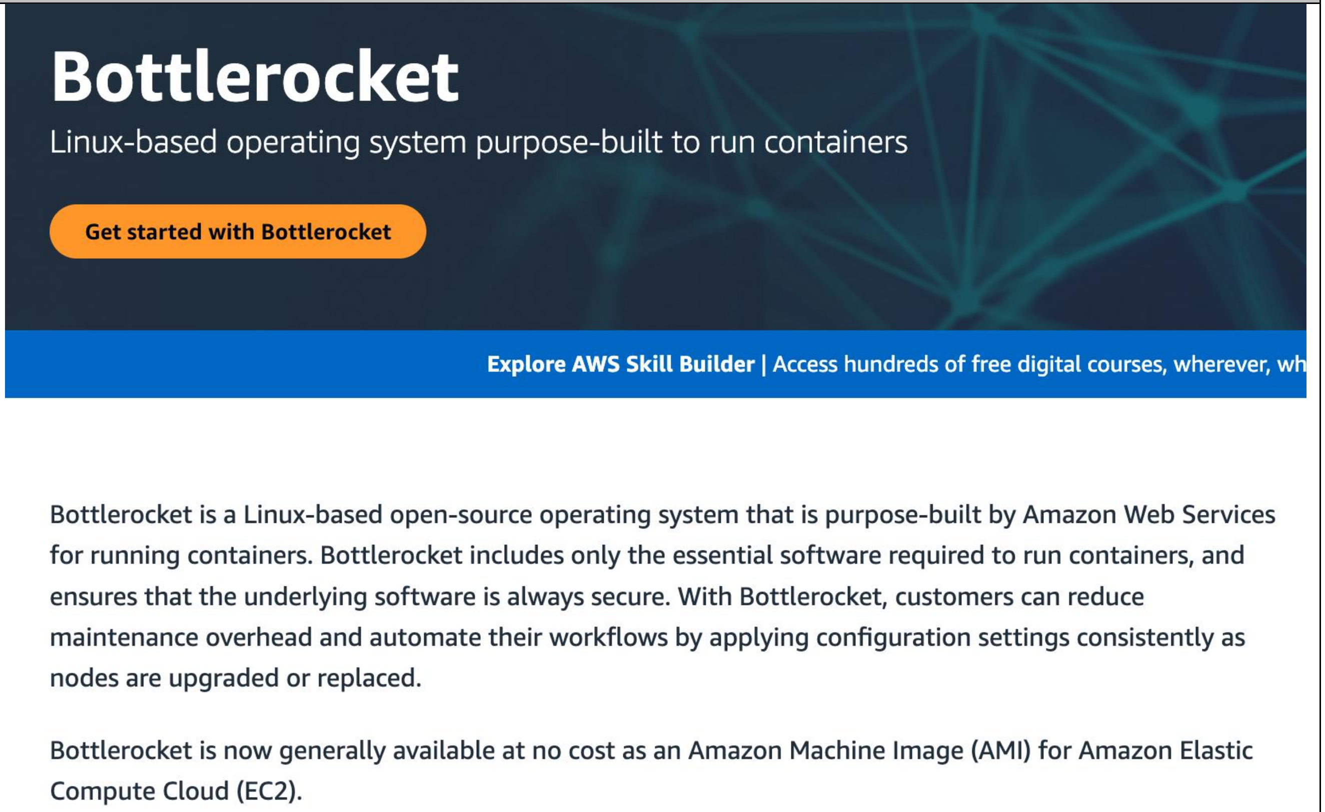
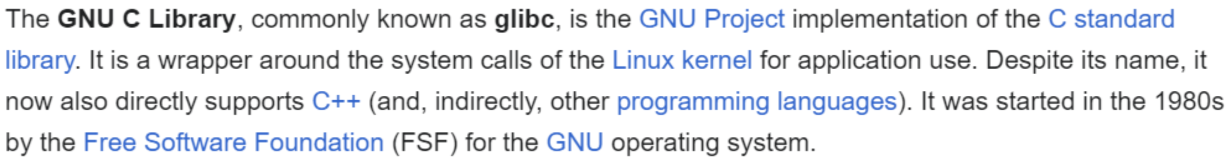
PDF | RSS

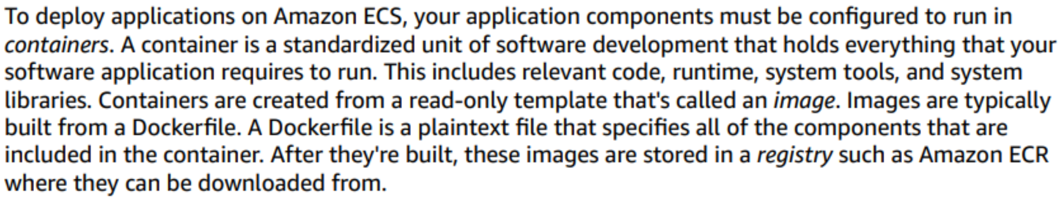
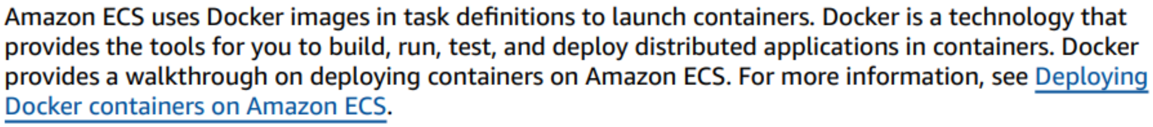
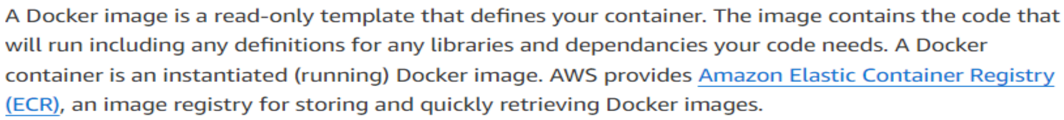
Amazon ECS Anywhere provides support for registering an *external instance* such as an on-premises server or virtual machine (VM), to your Amazon ECS cluster. External instances are optimized for running applications that generate outbound traffic or process data. If your application requires inbound traffic, the lack of Elastic Load Balancing support makes running these workloads less efficient. Amazon ECS added a new `EXTERNAL` launch type that you can use to create services or run tasks on your external instances.

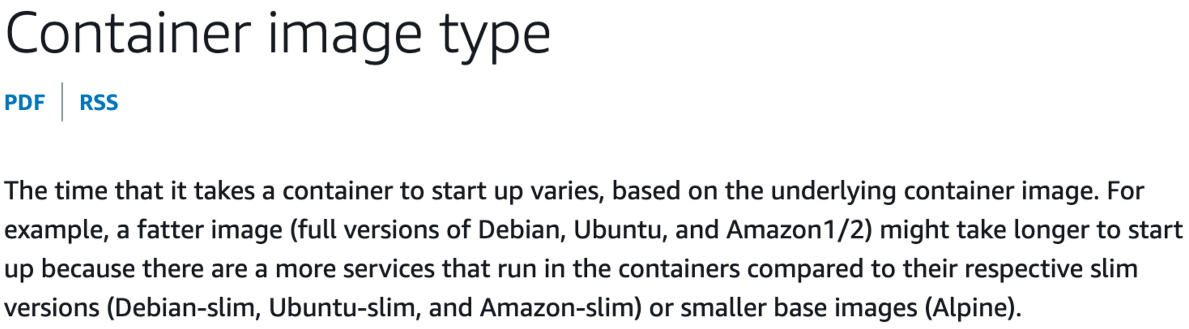
The following provides a high-level system architecture overview of Amazon ECS Anywhere. Your on-premises server has both the Amazon ECS agent and the SSM agent installed.

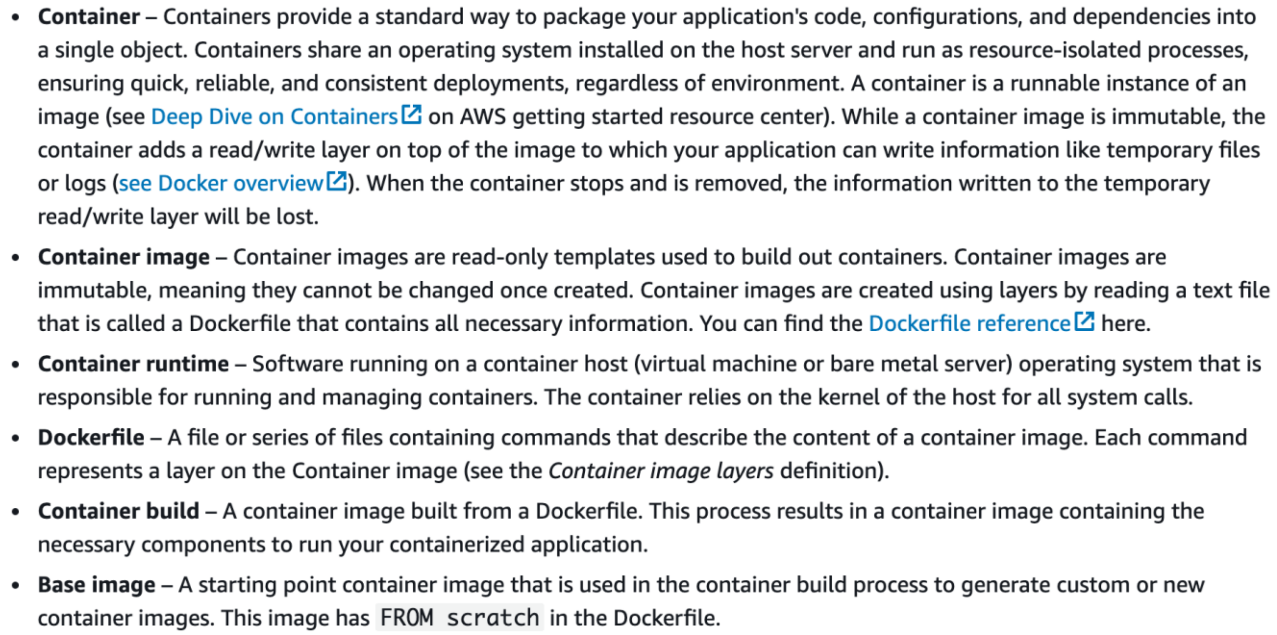
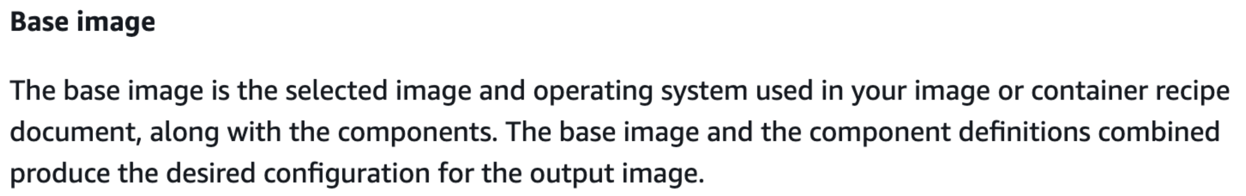
https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-anywhere.html

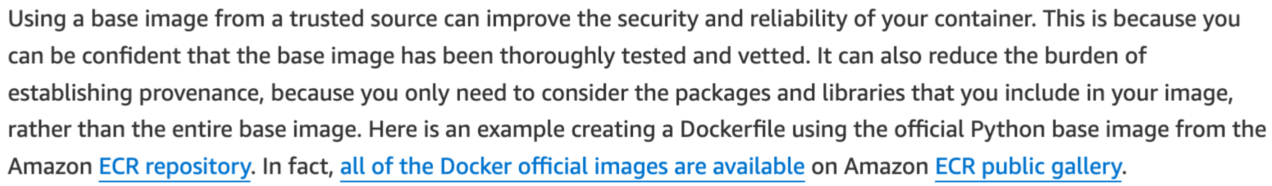
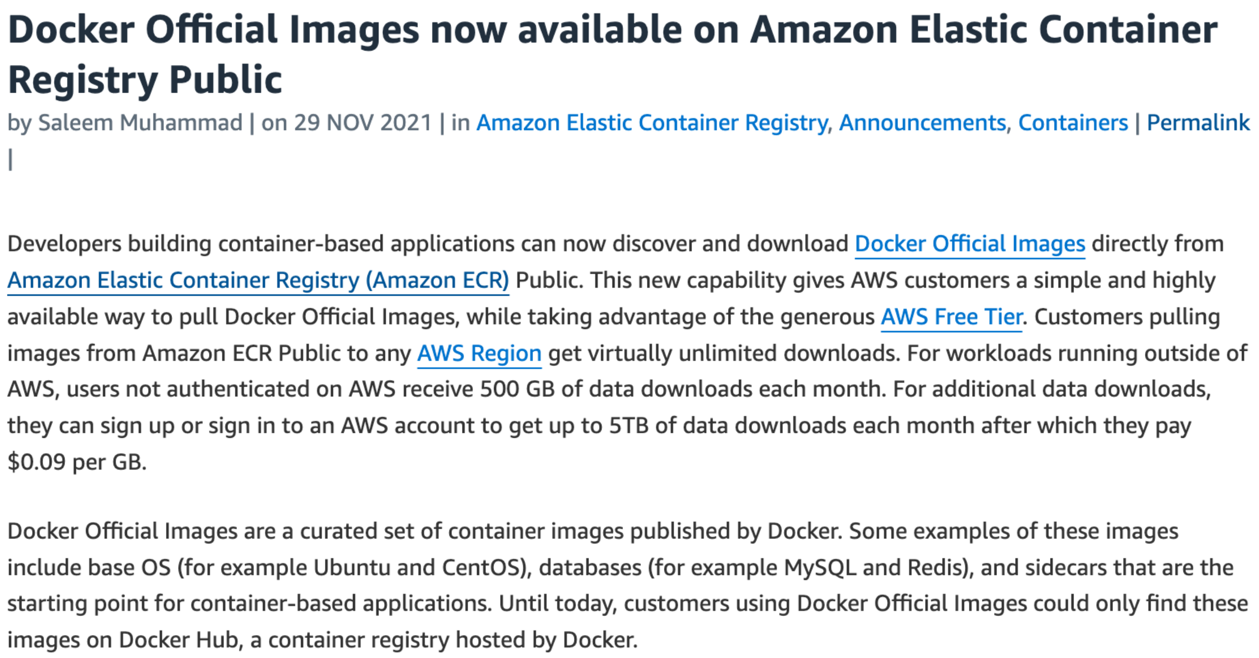
Claim 1	Accused Instrumentalities
	 **Supported operating systems and system architectures** The following is the list of supported operating systems and system architectures. • Amazon Linux 2 • CentOS 7 • RHEL 7, RHEL 8 — Neither Docker or RHEL's open package repositories support installing Docker natively on RHEL. You must ensure that Docker is installed before you run the install script that's described in this document. • Fedora 32, Fedora 33 • openSUSE Tumbleweed • Ubuntu 18, Ubuntu 20, Ubuntu 22 • Debian 10 ⚠ **Important** Debian 9 Long Term Support (LTS support) ended on June 30, 2022 and is no longer supported by Amazon ECS Anywhere. • Debian 11 • Debian 12 — The NVIDIA Container Toolkit isn't currently supported on Debian 12. You won't be able to run GPUs on Debian 12 instances. • SUSE Enterprise Server 15 • The `x86_64` and `ARM64` CPU architectures are supported. • The following Windows operating system versions are supported: ◦ Windows Server 2022 ◦ Windows Server 2019 ◦ Windows Server 2016 ◦ Windows Server 20H2 https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-anywhere.html

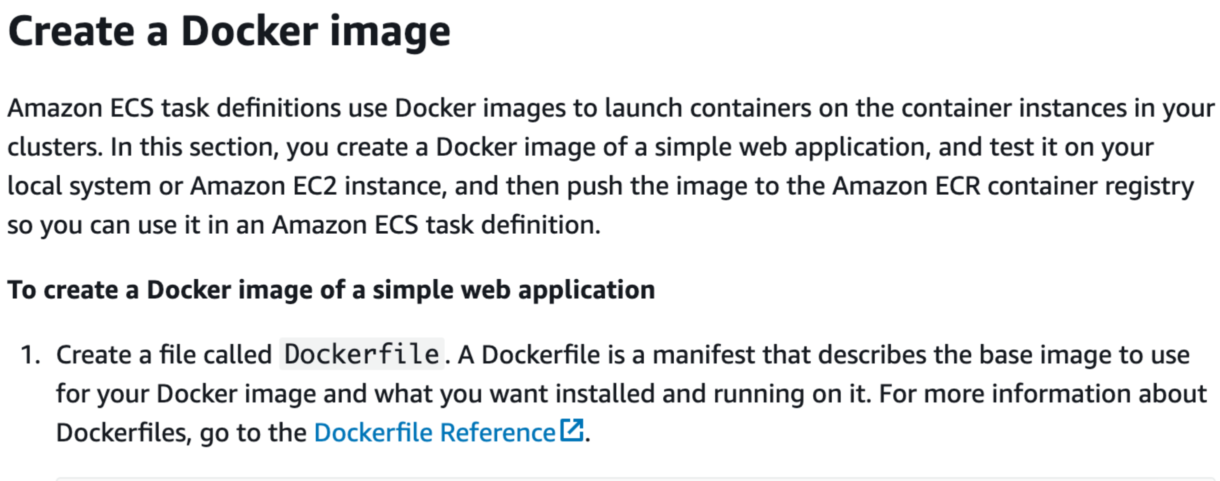
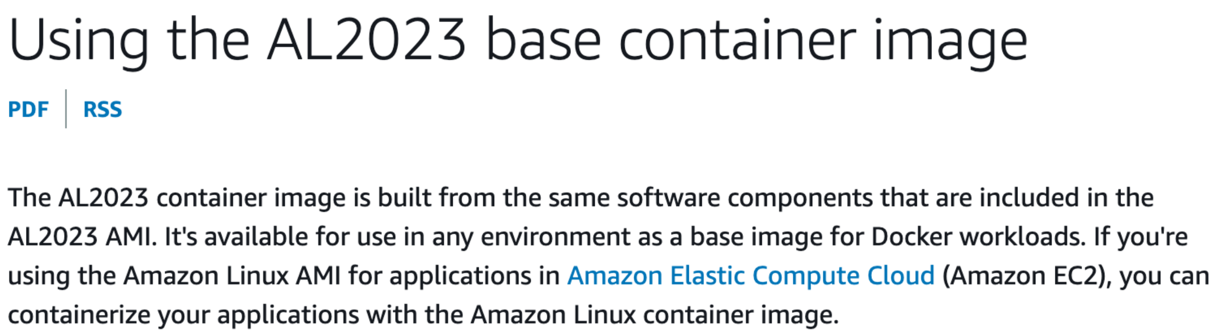
Claim 1	Accused Instrumentalities
	 Bottlerocket Linux-based operating system purpose-built to run containers Get started with Bottlerocket **Explore AWS Skill Builder** | Access hundreds of free digital courses, wherever, wh Bottlerocket is a Linux-based open-source operating system that is purpose-built by Amazon Web Services for running containers. Bottlerocket includes only the essential software required to run containers, and ensures that the underlying software is always secure. With Bottlerocket, customers can reduce maintenance overhead and automate their workflows by applying configuration settings consistently as nodes are upgraded or replaced. Bottlerocket is now generally available at no cost as an Amazon Machine Image (AMI) for Amazon Elastic Compute Cloud (EC2). https://aws.amazon.com/bottlerocket/  The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc

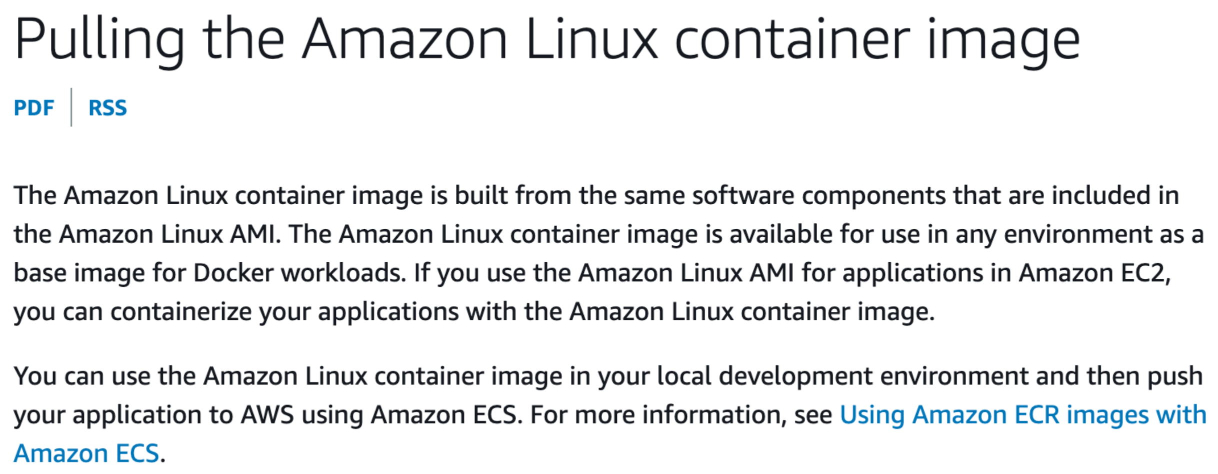
Claim 1	Accused Instrumentalities
[1c] c) a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode and	Each Accused Instrumentality comprises a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode. *See, e.g.*:  To deploy applications on Amazon ECS, your application components must be configured to run in *containers*. A container is a standardized unit of software development that holds everything that your software application requires to run. This includes relevant code, runtime, system tools, and system libraries. Containers are created from a read-only template that's called an *image*. Images are typically built from a Dockerfile. A Dockerfile is a plaintext file that specifies all of the components that are included in the container. After they're built, these images are stored in a *registry* such as Amazon ECR where they can be downloaded from. https://docs.aws.amazon.com/pdfs/AmazonECS/latest/developerguide/ecs-dg.pdf  Amazon ECS uses Docker images in task definitions to launch containers. Docker is a technology that provides the tools for you to build, run, test, and deploy distributed applications in containers. Docker provides a walkthrough on deploying containers on Amazon ECS. For more information, see Deploying Docker containers on Amazon ECS. https://docs.aws.amazon.com/pdfs/AmazonECS/latest/developerguide/ecs-dg.pdf  A Docker image is a read-only template that defines your container. The image contains the code that will run including any definitions for any libraries and dependancies your code needs. A Docker container is an instantiated (running) Docker image. AWS provides Amazon Elastic Container Registry (ECR), an image registry for storing and quickly retrieving Docker images. https://aws.amazon.com/docker/#

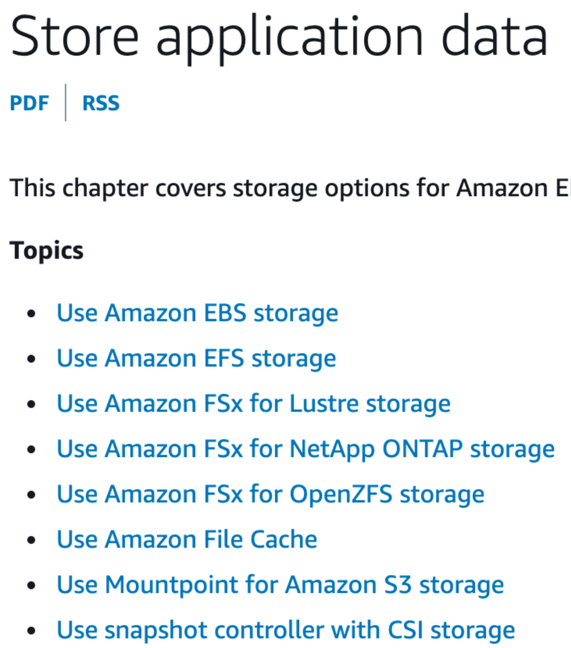
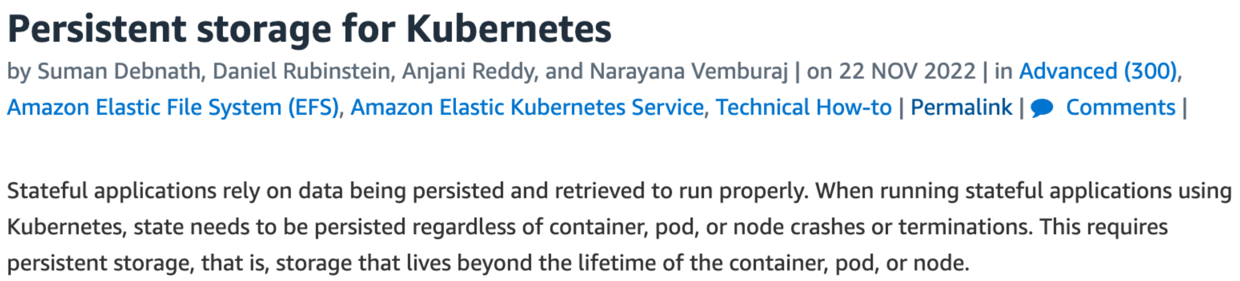
Claim 1	Accused Instrumentalities
	 Container image type PDF RSS The time that it takes a container to start up varies, based on the underlying container image. For example, a fatter image (full versions of Debian, Ubuntu, and Amazon1/2) might take longer to start up because there are a more services that run in the containers compared to their respective slim versions (Debian-slim, Ubuntu-slim, and Amazon-slim) or smaller base images (Alpine). https://docs.aws.amazon.com/AmazonECS/latest/bestpracticesguide/container-type.html

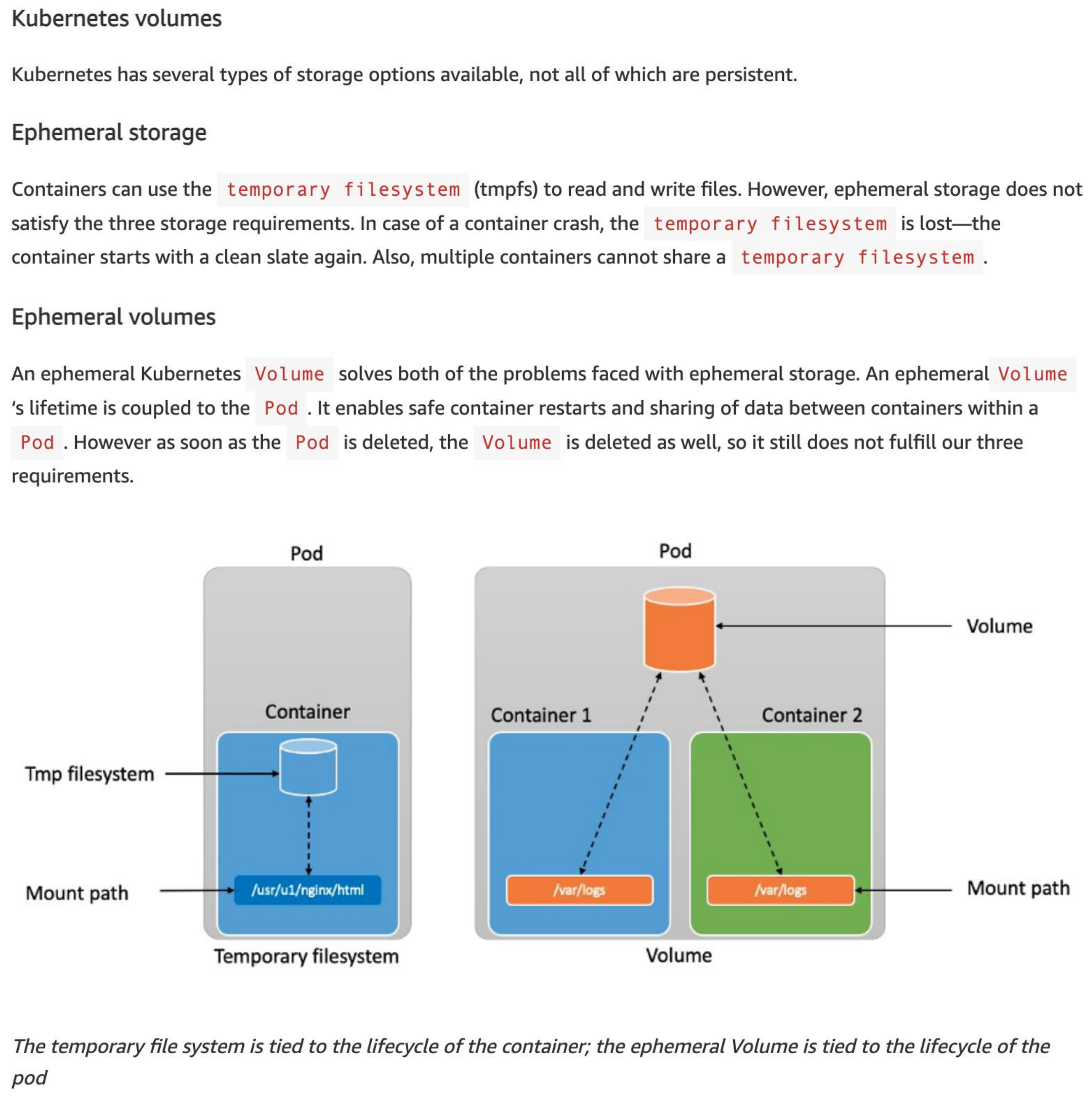
Claim 1	Accused Instrumentalities
	 • **Container** – Containers provide a standard way to package your application's code, configurations, and dependencies into a single object. Containers share an operating system installed on the host server and run as resource-isolated processes, ensuring quick, reliable, and consistent deployments, regardless of environment. A container is a runnable instance of an image (see Deep Dive on Containers on AWS getting started resource center). While a container image is immutable, the container adds a read/write layer on top of the image to which your application can write information like temporary files or logs (see Docker overview). When the container stops and is removed, the information written to the temporary read/write layer will be lost. • **Container image** – Container images are read-only templates used to build out containers. Container images are immutable, meaning they cannot be changed once created. Container images are created using layers by reading a text file that is called a Dockerfile that contains all necessary information. You can find the Dockerfile reference here. • **Container runtime** – Software running on a container host (virtual machine or bare metal server) operating system that is responsible for running and managing containers. The container relies on the kernel of the host for all system calls. • **Dockerfile** – A file or series of files containing commands that describe the content of a container image. Each command represents a layer on the Container image (see the *Container image layers* definition). • **Container build** – A container image built from a Dockerfile. This process results in a container image containing the necessary components to run your containerized application. • **Base image** – A starting point container image that is used in the container build process to generate custom or new container images. This image has `FROM scratch` in the Dockerfile. https://docs.aws.amazon.com/wellarchitected/latest/container-build-lens/container-technology-terminology.html  **Base image** The base image is the selected image and operating system used in your image or container recipe document, along with the components. The base image and the component definitions combined produce the desired configuration for the output image. https://docs.aws.amazon.com/imagebuilder/latest/userguide/what-is-image-builder.html

Claim 1	Accused Instrumentalities
	 Using a base image from a trusted source can improve the security and reliability of your container. This is because you can be confident that the base image has been thoroughly tested and vetted. It can also reduce the burden of establishing provenance, because you only need to consider the packages and libraries that you include in your image, rather than the entire base image. Here is an example creating a Dockerfile using the official Python base image from the Amazon ECR repository. In fact, all of the Docker official images are available on Amazon ECR public gallery. https://aws.amazon.com/blogs/containers/building-better-container-images/  **Docker Official Images now available on Amazon Elastic Container Registry Public** by Saleem Muhammad | on 29 NOV 2021 | in Amazon Elastic Container Registry, Announcements, Containers | Permalink | Developers building container-based applications can now discover and download Docker Official Images directly from Amazon Elastic Container Registry (Amazon ECR) Public. This new capability gives AWS customers a simple and highly available way to pull Docker Official Images, while taking advantage of the generous AWS Free Tier. Customers pulling images from Amazon ECR Public to any AWS Region get virtually unlimited downloads. For workloads running outside of AWS, users not authenticated on AWS receive 500 GB of data downloads each month. For additional data downloads, they can sign up or sign in to an AWS account to get up to 5TB of data downloads each month after which they pay $0.09 per GB. Docker Official Images are a curated set of container images published by Docker. Some examples of these images include base OS (for example Ubuntu and CentOS), databases (for example MySQL and Redis), and sidecars that are the starting point for container-based applications. Until today, customers using Docker Official Images could only find these images on Docker Hub, a container registry hosted by Docker. https://aws.amazon.com/blogs/containers/docker-official-images-now-available-on-amazon-elastic-container-registry-public/

Claim 1	Accused Instrumentalities
	 **Create a Docker image** Amazon ECS task definitions use Docker images to launch containers on the container instances in your clusters. In this section, you create a Docker image of a simple web application, and test it on your local system or Amazon EC2 instance, and then push the image to the Amazon ECR container registry so you can use it in an Amazon ECS task definition. **To create a Docker image of a simple web application** 1. Create a file called `Dockerfile`. A Dockerfile is a manifest that describes the base image to use for your Docker image and what you want installed and running on it. For more information about Dockerfiles, go to the Dockerfile Reference. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create-container-image.html  **Using the AL2023 base container image** The AL2023 container image is built from the same software components that are included in the AL2023 AMI. It's available for use in any environment as a base image for Docker workloads. If you're using the Amazon Linux AMI for applications in Amazon Elastic Compute Cloud (Amazon EC2), you can containerize your applications with the Amazon Linux container image. https://docs.aws.amazon.com/linux/al2023/ug/base-container.html

Claim 1	Accused Instrumentalities
	 Pulling the Amazon Linux container image PDF | RSS The Amazon Linux container image is built from the same software components that are included in the Amazon Linux AMI. The Amazon Linux container image is available for use in any environment as a base image for Docker workloads. If you use the Amazon Linux AMI for applications in Amazon EC2, you can containerize your applications with the Amazon Linux container image. You can use the Amazon Linux container image in your local development environment and then push your application to AWS using Amazon ECS. For more information, see Using Amazon ECR images with Amazon ECS. https://docs.aws.amazon.com/AmazonECR/latest/userguide/amazon_linux_container_image.html

Claim 1	Accused Instrumentalities
	 Store application data PDF | RSS This chapter covers storage options for Amazon EKS clusters. **Topics** - Use Amazon EBS storage - Use Amazon EFS storage - Use Amazon FSx for Lustre storage - Use Amazon FSx for NetApp ONTAP storage - Use Amazon FSx for OpenZFS storage - Use Amazon File Cache - Use Mountpoint for Amazon S3 storage - Use snapshot controller with CSI storage https://docs.aws.amazon.com/eks/latest/userguide/storage.html  **Persistent storage for Kubernetes** by Suman Debnath, Daniel Rubinstein, Anjani Reddy, and Narayana Vemburaj | on 22 NOV 2022 | in Advanced (300), Amazon Elastic File System (EFS), Amazon Elastic Kubernetes Service, Technical How-to | Permalink | Comments | Stateful applications rely on data being persisted and retrieved to run properly. When running stateful applications using Kubernetes, state needs to be persisted regardless of container, pod, or node crashes or terminations. This requires persistent storage, that is, storage that lives beyond the lifetime of the container, pod, or node. https://aws.amazon.com/blogs/storage/persistent-storage-for-kubernetes/



Kubernetes volumes

Kubernetes has several types of storage options available, not all of which are persistent.

Ephemeral storage

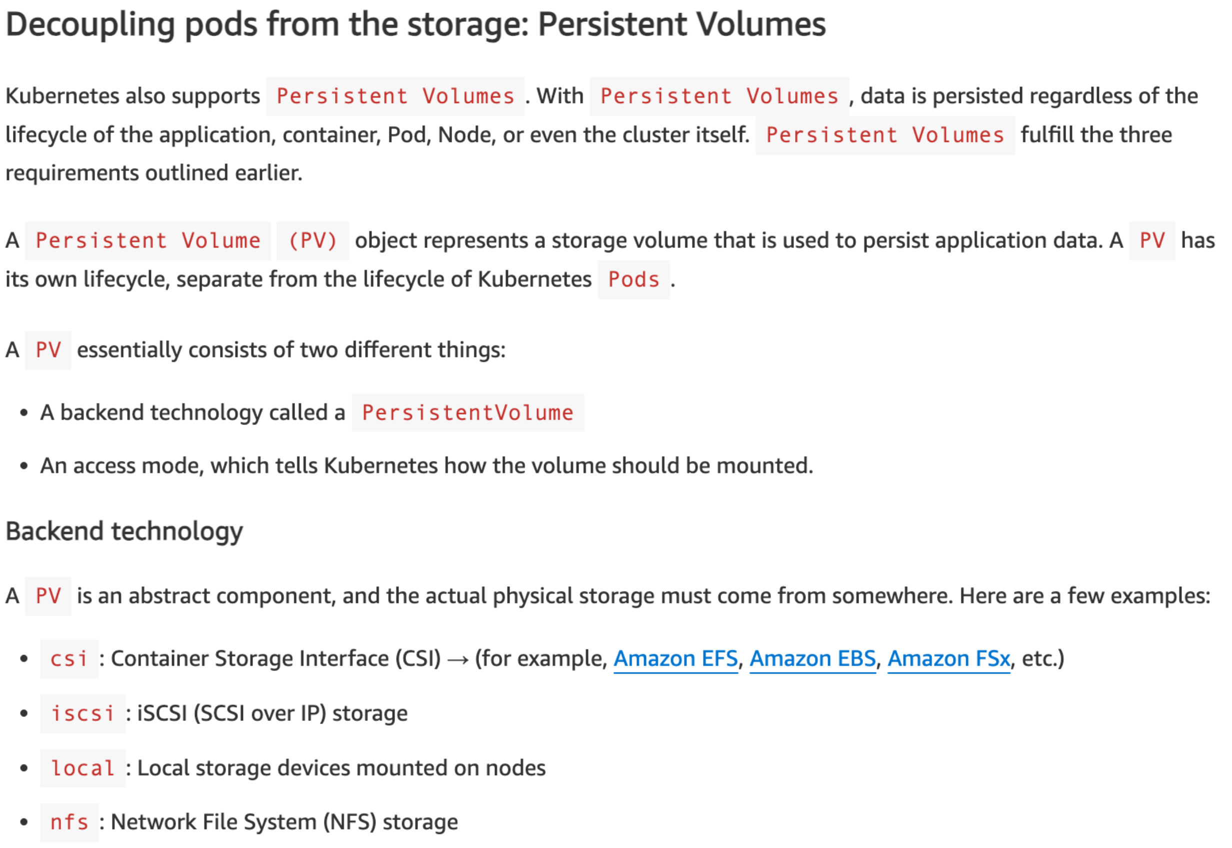
Containers can use the `temporary filesystem` (tmpfs) to read and write files. However, ephemeral storage does not satisfy the three storage requirements. In case of a container crash, the `temporary filesystem` is lost—the container starts with a clean slate again. Also, multiple containers cannot share a `temporary filesystem` .

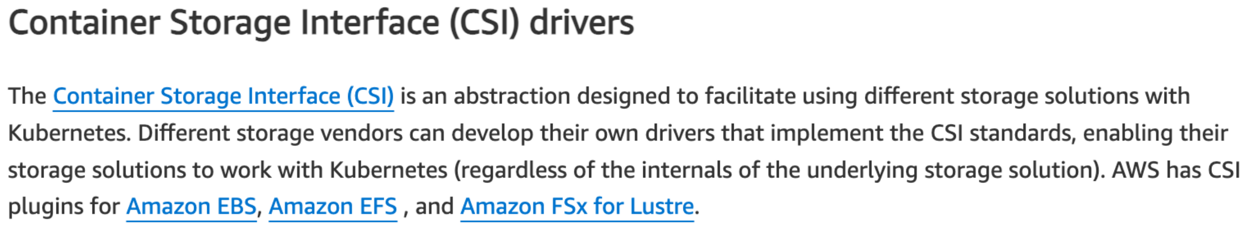
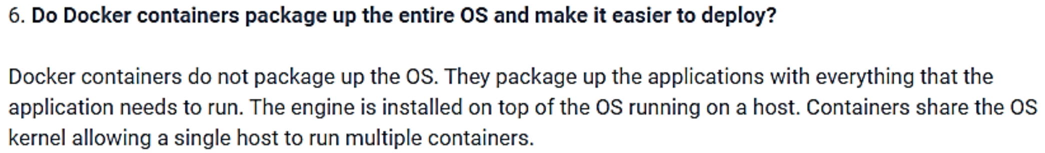
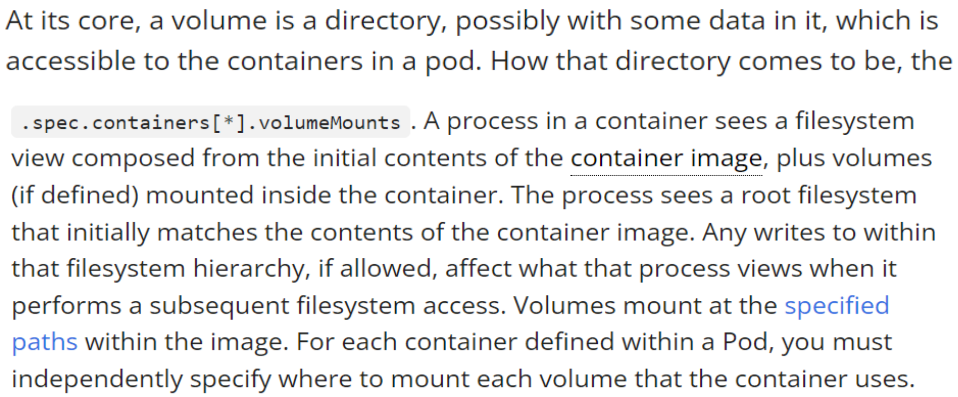
Ephemeral volumes

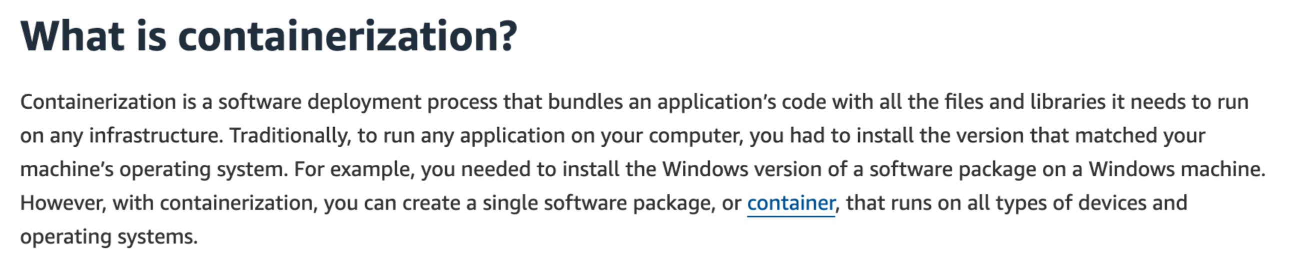
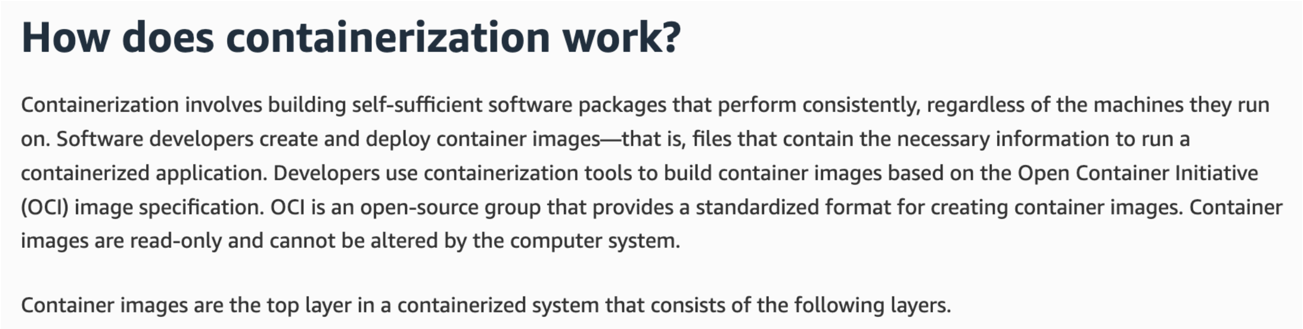
An ephemeral Kubernetes `Volume` solves both of the problems faced with ephemeral storage. An ephemeral `Volume` 's lifetime is coupled to the `Pod` . It enables safe container restarts and sharing of data between containers within a `Pod` . However as soon as the `Pod` is deleted, the `Volume` is deleted as well, so it still does not fulfill our three requirements.

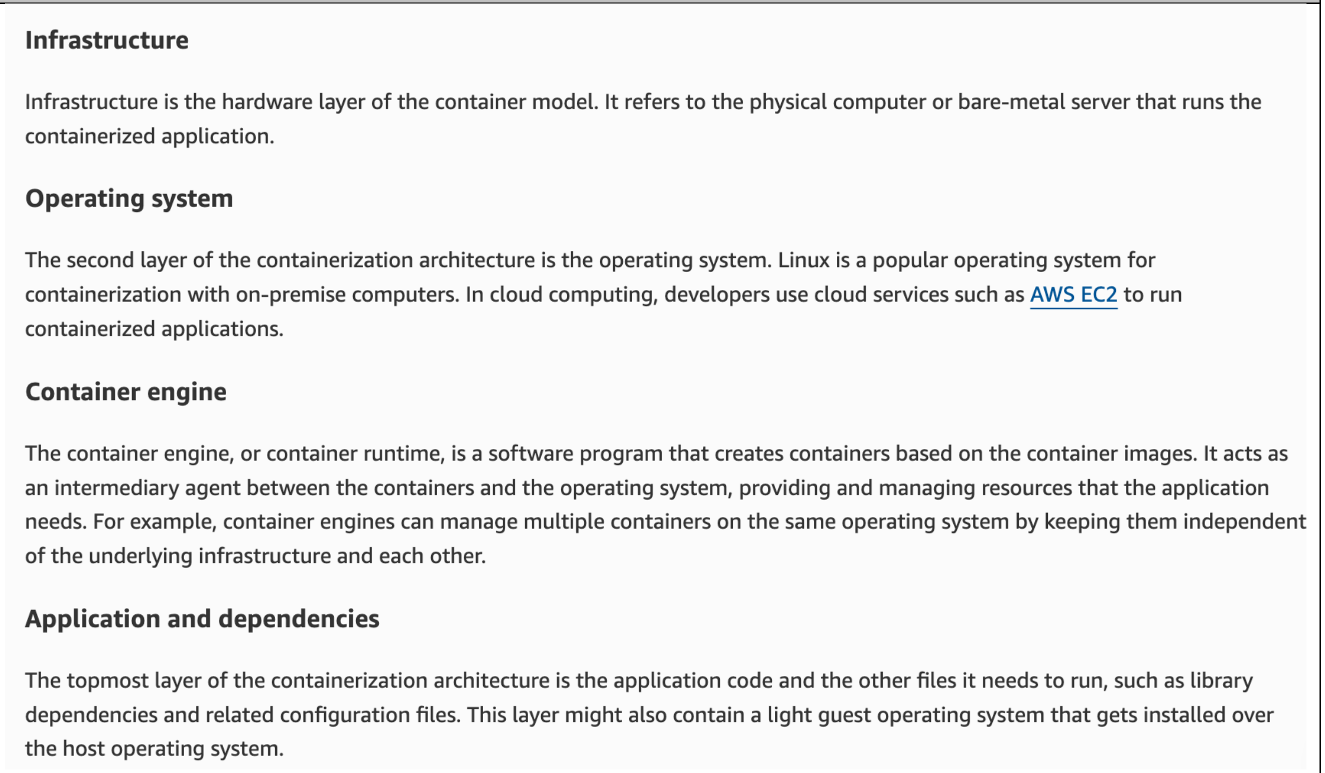
The temporary file system is tied to the lifecycle of the container; the ephemeral Volume is tied to the lifecycle of the pod

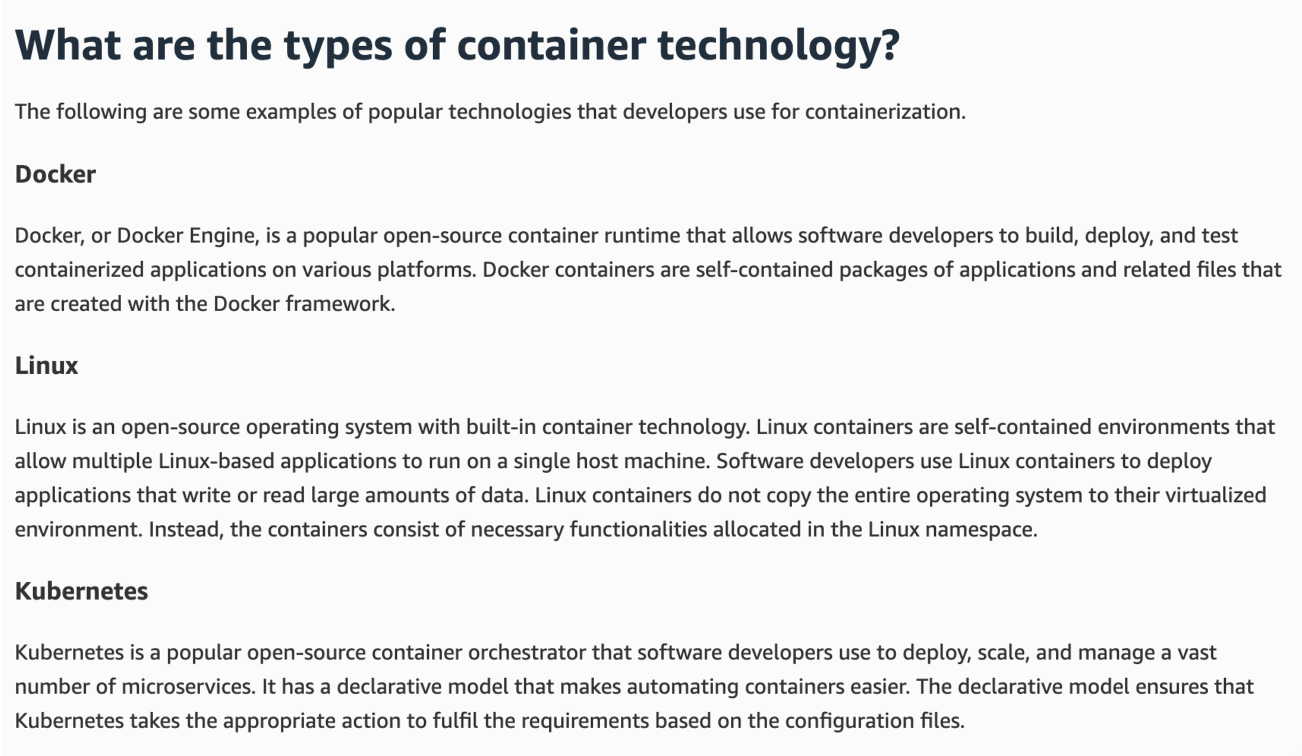
https://aws.amazon.com/blogs/storage/persistent-storage-for-kubernetes/

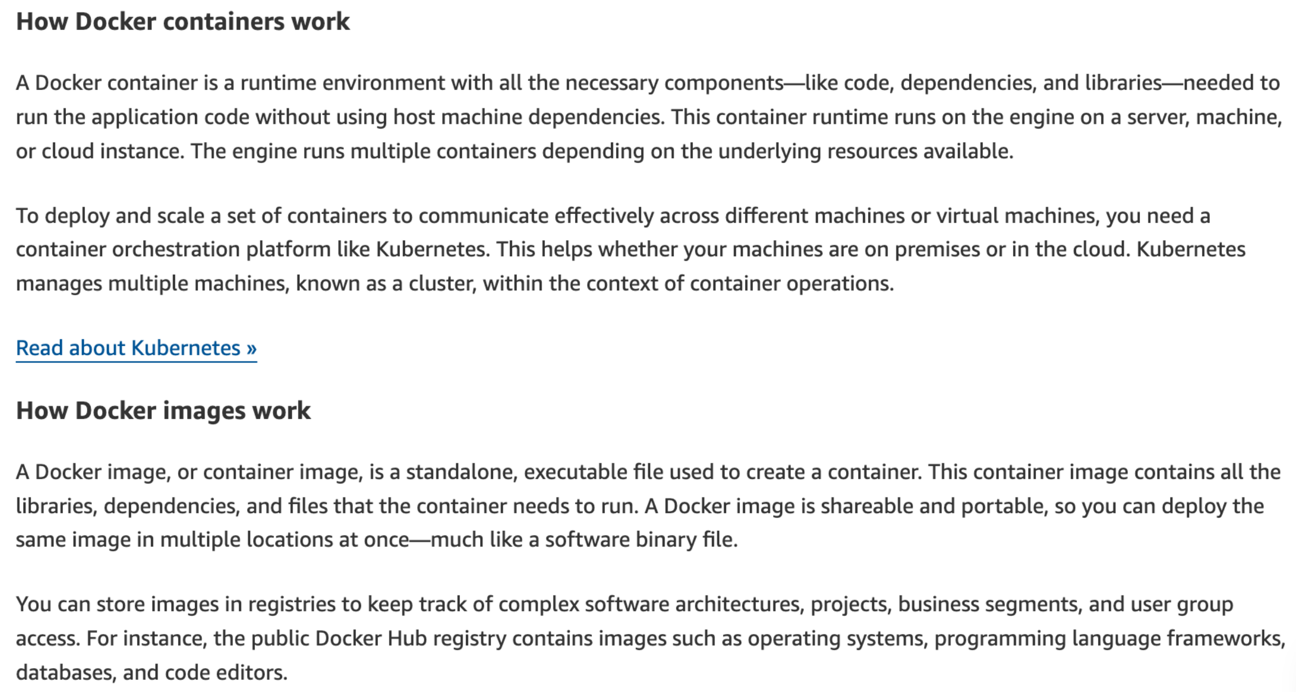
Claim 1	Accused Instrumentalities
	 **Decoupling pods from the storage: Persistent Volumes** Kubernetes also supports `Persistent Volumes`. With `Persistent Volumes`, data is persisted regardless of the lifecycle of the application, container, Pod, Node, or even the cluster itself. `Persistent Volumes` fulfill the three requirements outlined earlier. A `Persistent Volume` `(PV)` object represents a storage volume that is used to persist application data. A `PV` has its own lifecycle, separate from the lifecycle of Kubernetes `Pods`. A `PV` essentially consists of two different things: • A backend technology called a `PersistentVolume` • An access mode, which tells Kubernetes how the volume should be mounted. **Backend technology** A `PV` is an abstract component, and the actual physical storage must come from somewhere. Here are a few examples: • `csi`: Container Storage Interface (CSI) → (for example, Amazon EFS, Amazon EBS, Amazon FSx, etc.) • `iscsi`: iSCSI (SCSI over IP) storage • `local`: Local storage devices mounted on nodes • `nfs`: Network File System (NFS) storage https://aws.amazon.com/blogs/storage/persistent-storage-for-kubernetes/

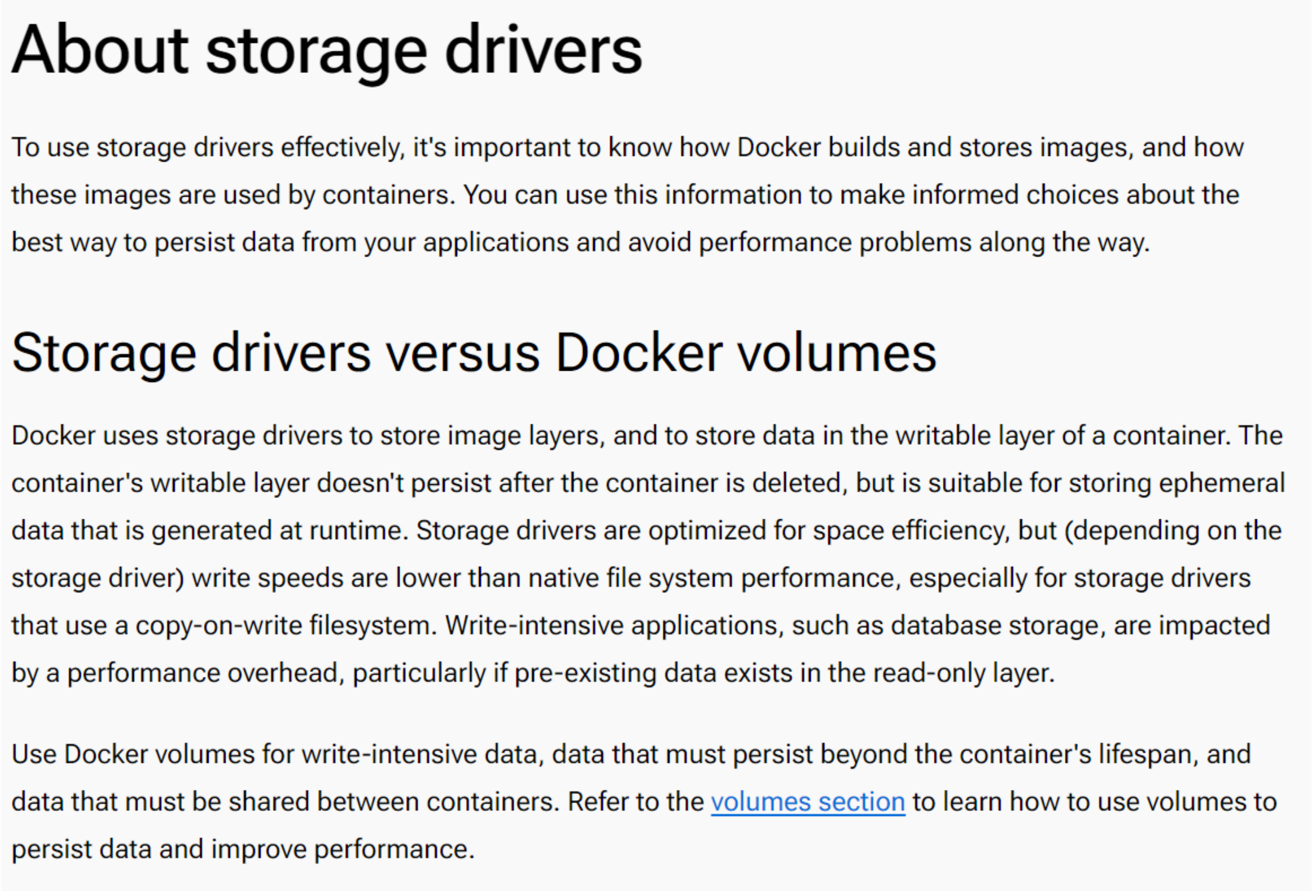
Claim 1	Accused Instrumentalities
	 **Container Storage Interface (CSI) drivers** The Container Storage Interface (CSI) is an abstraction designed to facilitate using different storage solutions with Kubernetes. Different storage vendors can develop their own drivers that implement the CSI standards, enabling their storage solutions to work with Kubernetes (regardless of the internals of the underlying storage solution). AWS has CSI plugins for Amazon EBS, Amazon EFS , and Amazon FSx for Lustre. https://aws.amazon.com/blogs/storage/persistent-storage-for-kubernetes/  6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/  At its core, a volume is a directory, possibly with some data in it, which is accessible to the containers in a pod. How that directory comes to be, the `.spec.containers[*].volumeMounts` . A process in a container sees a filesystem view composed from the initial contents of the container image, plus volumes (if defined) mounted inside the container. The process sees a root filesystem that initially matches the contents of the container image. Any writes to within that filesystem hierarchy, if allowed, affect what that process views when it performs a subsequent filesystem access. Volumes mount at the specified paths within the image. For each container defined within a Pod, you must independently specify where to mount each volume that the container uses. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	 **What is containerization?** Containerization is a software deployment process that bundles an application's code with all the files and libraries it needs to run on any infrastructure. Traditionally, to run any application on your computer, you had to install the version that matched your machine's operating system. For example, you needed to install the Windows version of a software package on a Windows machine. However, with containerization, you can create a single software package, or container, that runs on all types of devices and operating systems. https://aws.amazon.com/what-is/containerization/  **How does containerization work?** Containerization involves building self-sufficient software packages that perform consistently, regardless of the machines they run on. Software developers create and deploy container images—that is, files that contain the necessary information to run a containerized application. Developers use containerization tools to build container images based on the Open Container Initiative (OCI) image specification. OCI is an open-source group that provides a standardized format for creating container images. Container images are read-only and cannot be altered by the computer system. Container images are the top layer in a containerized system that consists of the following layers. https://aws.amazon.com/what-is/containerization/

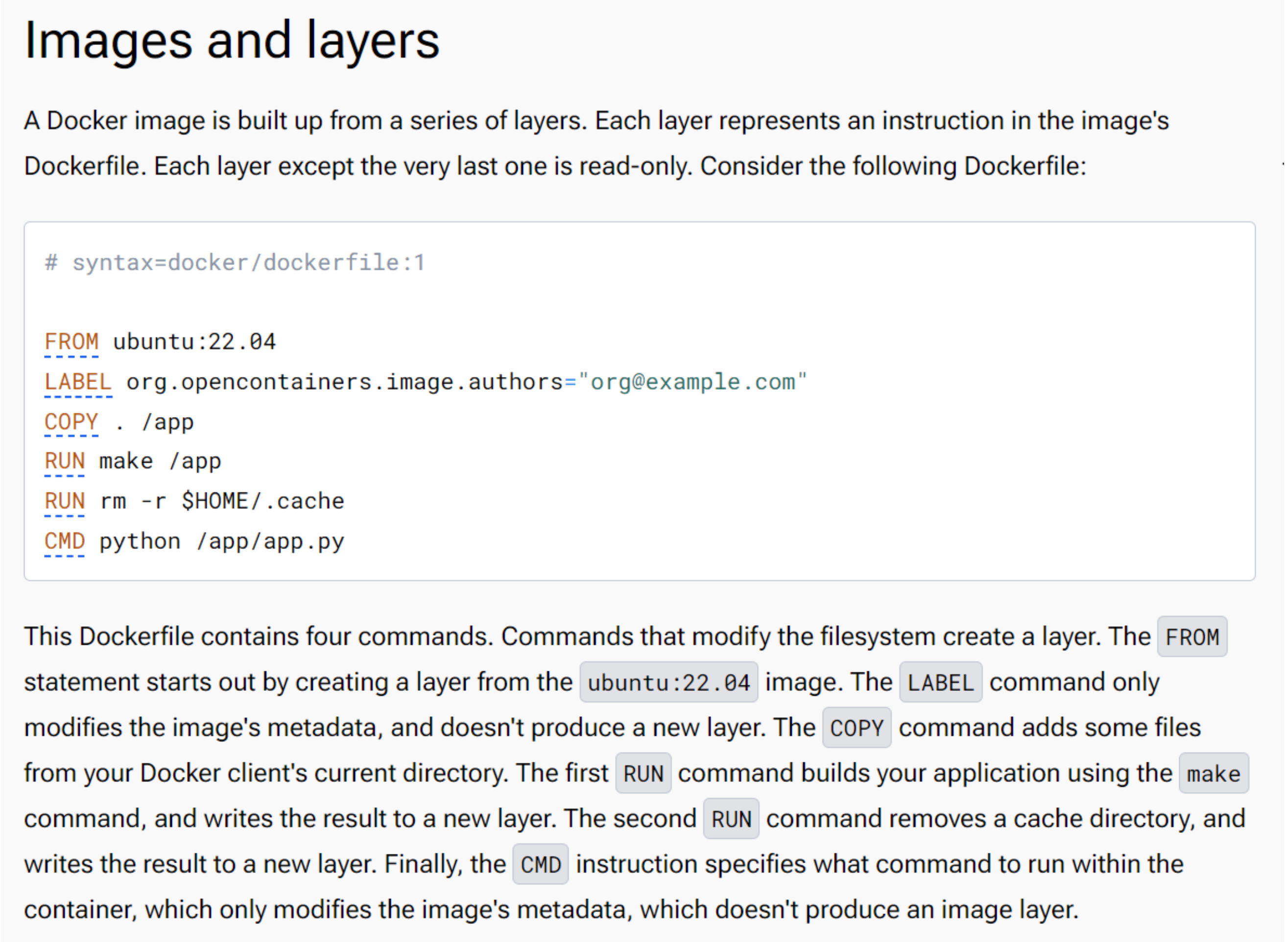
Claim 1	Accused Instrumentalities
	 **Infrastructure** Infrastructure is the hardware layer of the container model. It refers to the physical computer or bare-metal server that runs the containerized application. **Operating system** The second layer of the containerization architecture is the operating system. Linux is a popular operating system for containerization with on-premise computers. In cloud computing, developers use cloud services such as AWS EC2 to run containerized applications. **Container engine** The container engine, or container runtime, is a software program that creates containers based on the container images. It acts as an intermediary agent between the containers and the operating system, providing and managing resources that the application needs. For example, container engines can manage multiple containers on the same operating system by keeping them independent of the underlying infrastructure and each other. **Application and dependencies** The topmost layer of the containerization architecture is the application code and the other files it needs to run, such as library dependencies and related configuration files. This layer might also contain a light guest operating system that gets installed over the host operating system. https://aws.amazon.com/what-is/containerization/

Claim 1	Accused Instrumentalities
	 **What are the types of container technology?** The following are some examples of popular technologies that developers use for containerization. **Docker** Docker, or Docker Engine, is a popular open-source container runtime that allows software developers to build, deploy, and test containerized applications on various platforms. Docker containers are self-contained packages of applications and related files that are created with the Docker framework. **Linux** Linux is an open-source operating system with built-in container technology. Linux containers are self-contained environments that allow multiple Linux-based applications to run on a single host machine. Software developers use Linux containers to deploy applications that write or read large amounts of data. Linux containers do not copy the entire operating system to their virtualized environment. Instead, the containers consist of necessary functionalities allocated in the Linux namespace. **Kubernetes** Kubernetes is a popular open-source container orchestrator that software developers use to deploy, scale, and manage a vast number of microservices. It has a declarative model that makes automating containers easier. The declarative model ensures that Kubernetes takes the appropriate action to fulfil the requirements based on the configuration files. https://aws.amazon.com/what-is/containerization/

Claim 1	Accused Instrumentalities
	 **How Docker containers work** A Docker container is a runtime environment with all the necessary components—like code, dependencies, and libraries—needed to run the application code without using host machine dependencies. This container runtime runs on the engine on a server, machine, or cloud instance. The engine runs multiple containers depending on the underlying resources available. To deploy and scale a set of containers to communicate effectively across different machines or virtual machines, you need a container orchestration platform like Kubernetes. This helps whether your machines are on premises or in the cloud. Kubernetes manages multiple machines, known as a cluster, within the context of container operations. Read about Kubernetes » **How Docker images work** A Docker image, or container image, is a standalone, executable file used to create a container. This container image contains all the libraries, dependencies, and files that the container needs to run. A Docker image is shareable and portable, so you can deploy the same image in multiple locations at once—much like a software binary file. You can store images in registries to keep track of complex software architectures, projects, business segments, and user group access. For instance, the public Docker Hub registry contains images such as operating systems, programming language frameworks, databases, and code editors. https://aws.amazon.com/compare/the-difference-between-docker-images-and-containers/

Claim 1	Accused Instrumentalities
	 About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities



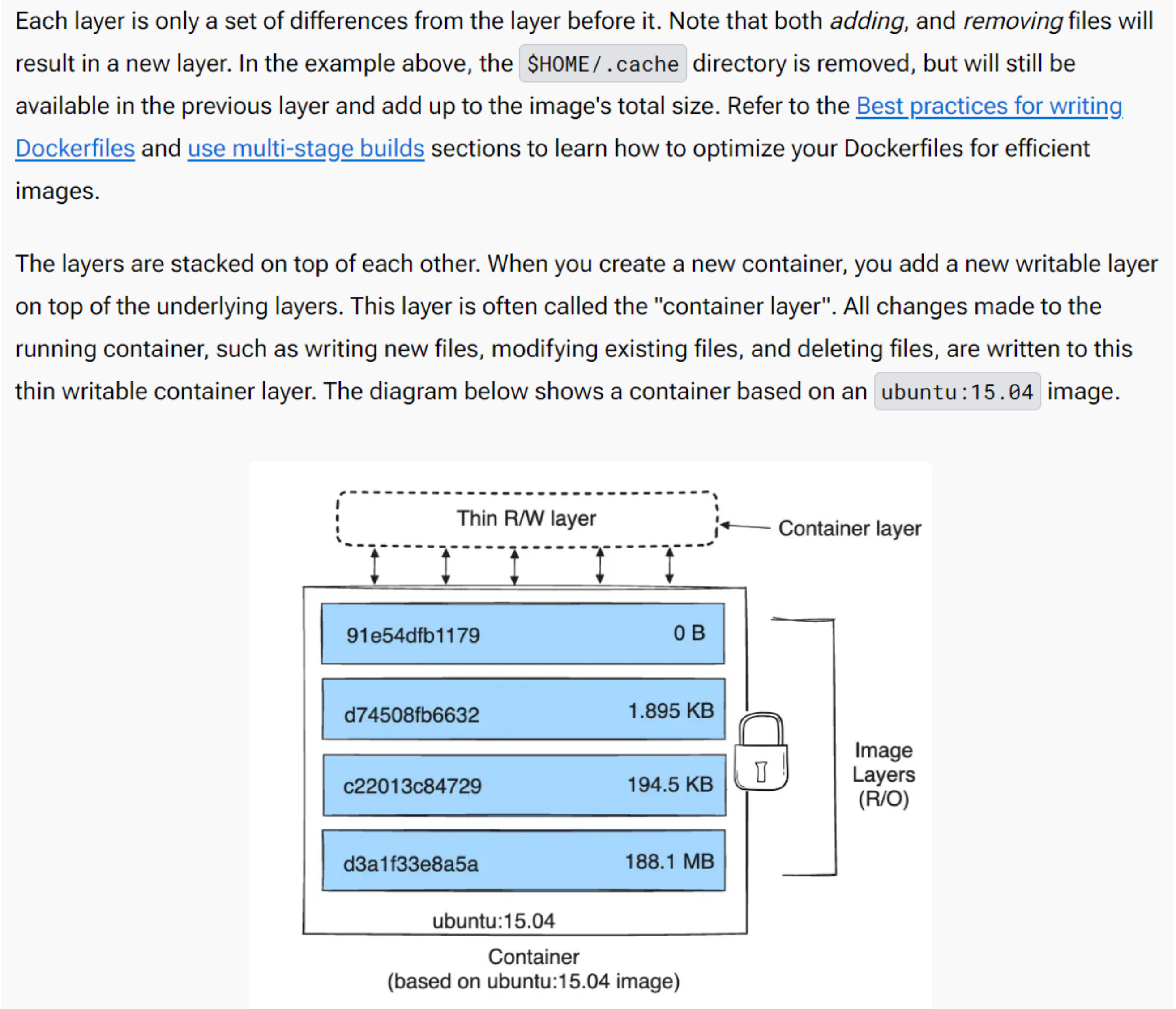
Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

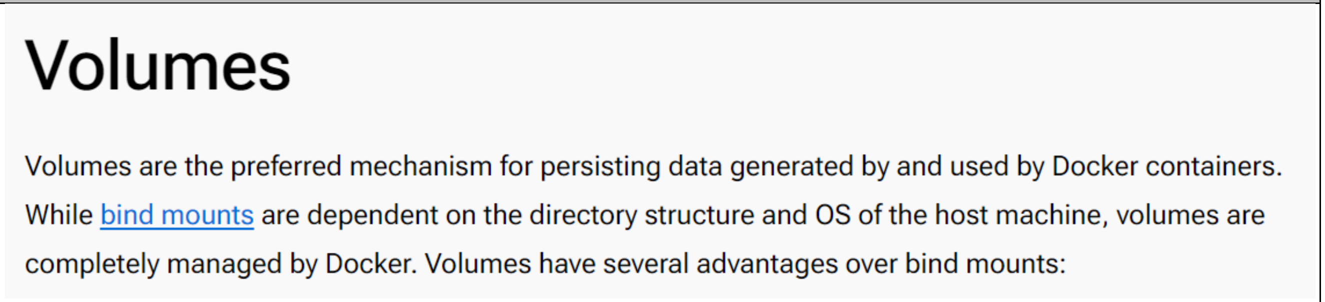
https://docs.docker.com/storage/storagedriver/



Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images.

The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.

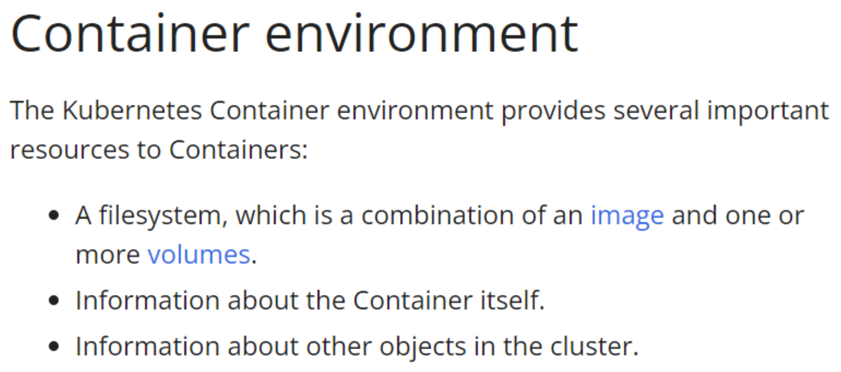
https://docs.docker.com/storage/storagedriver/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>


Volumes

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

https://kubernetes.io/docs/concepts/storage/volumes/

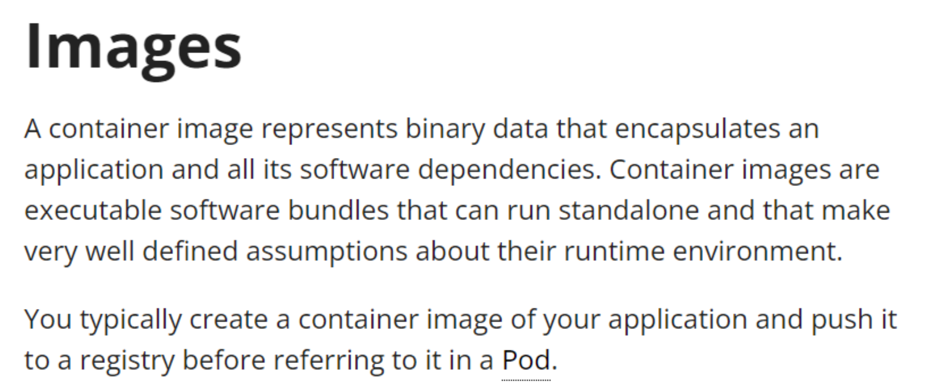
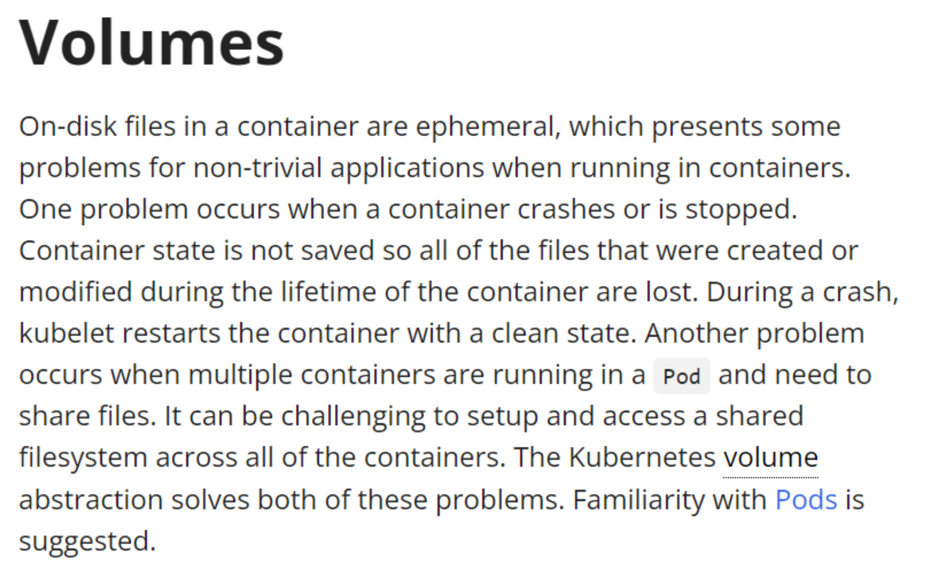


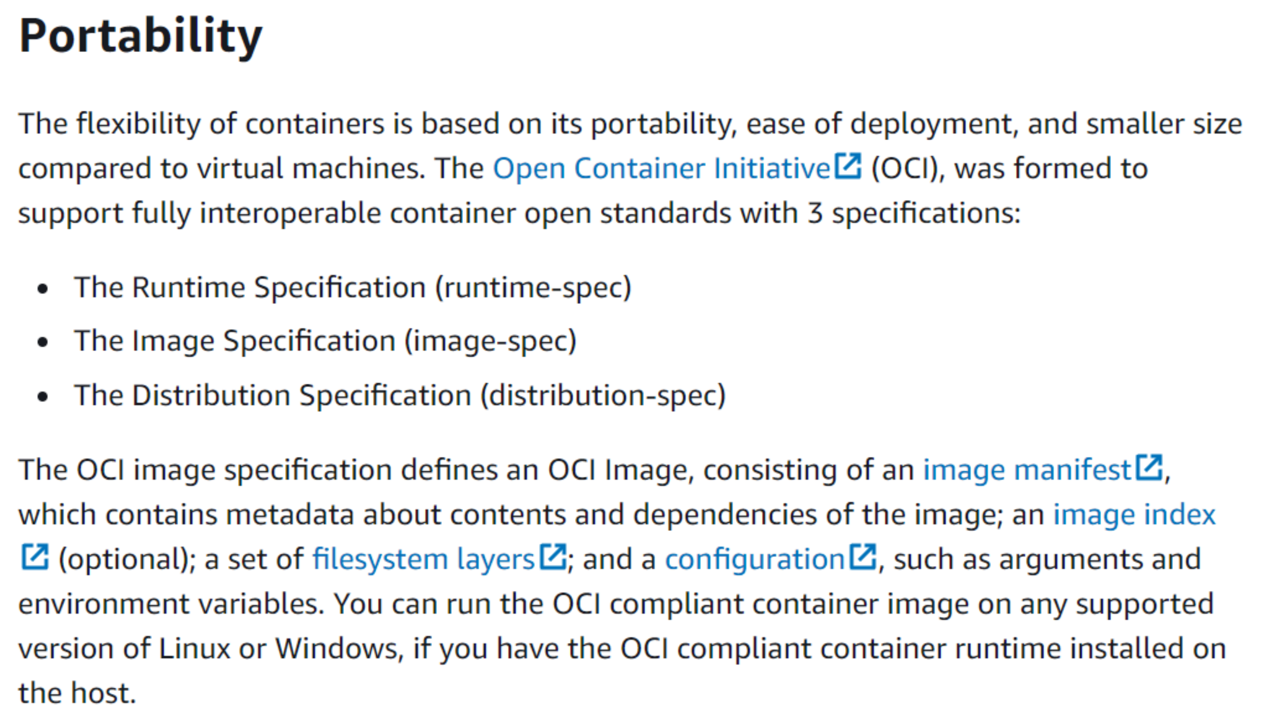
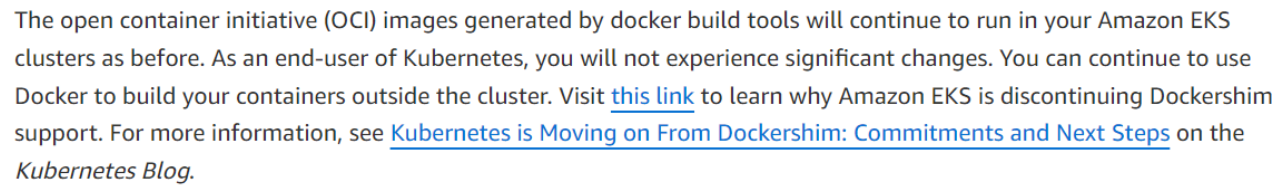
Container environment

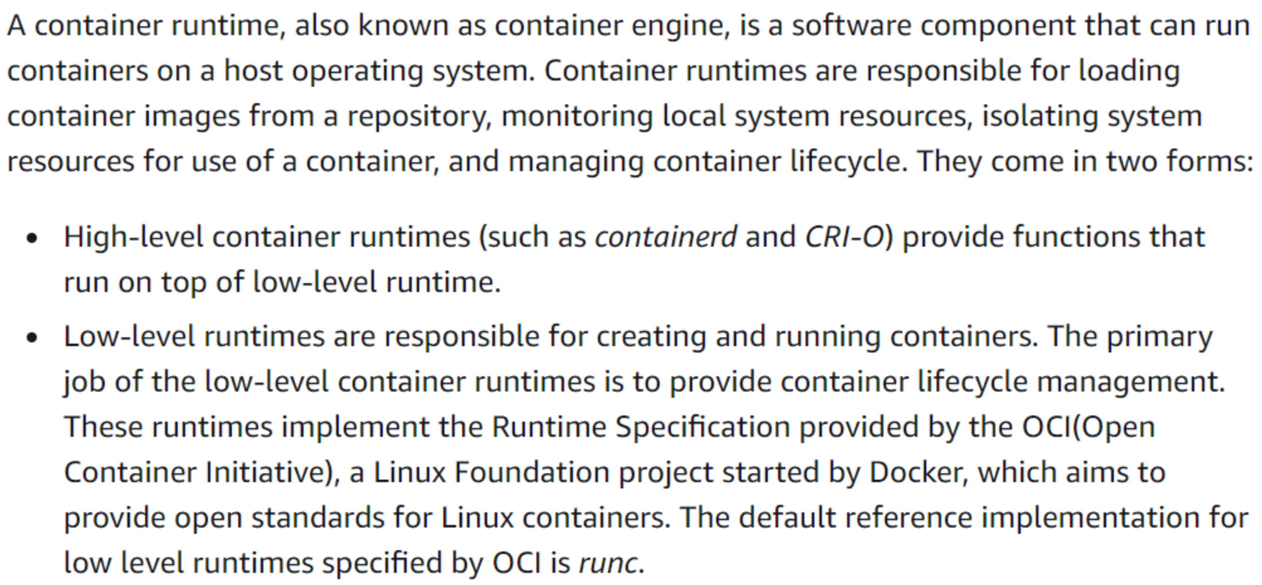
The Kubernetes Container environment provides several important resources to Containers:

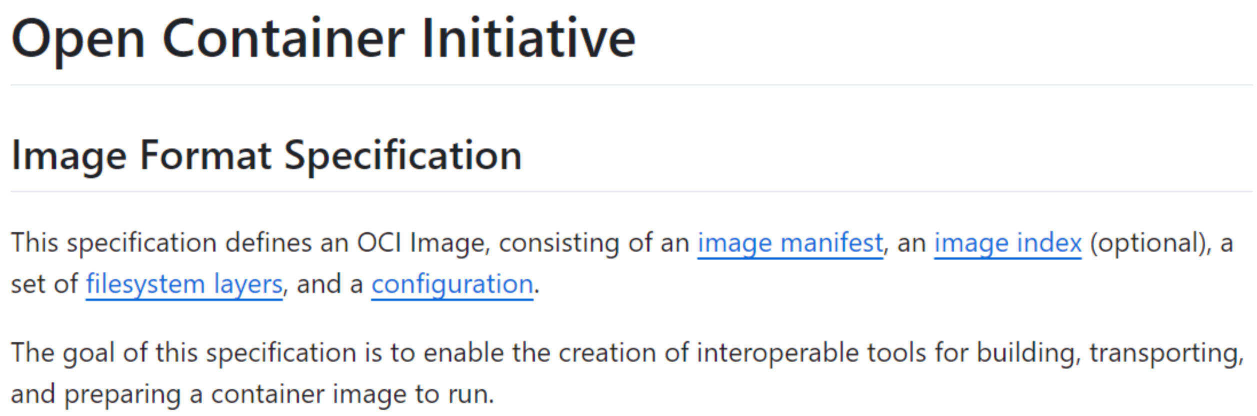
- A filesystem, which is a combination of an image and one or more volumes.
- Information about the Container itself.
- Information about other objects in the cluster.

https://kubernetes.io/docs/concepts/containers/container-environment/
</td></tr>
</table>

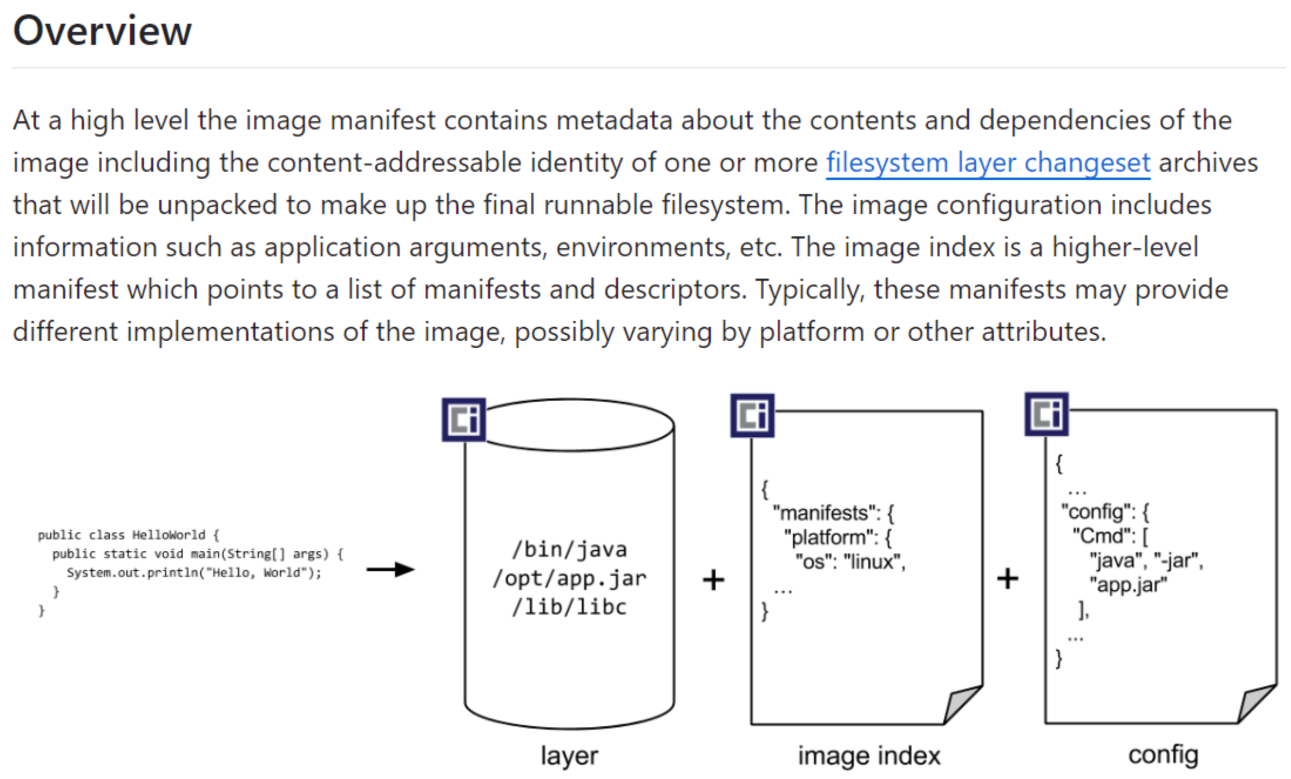
Claim 1	Accused Instrumentalities
	 **Images** A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/  **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

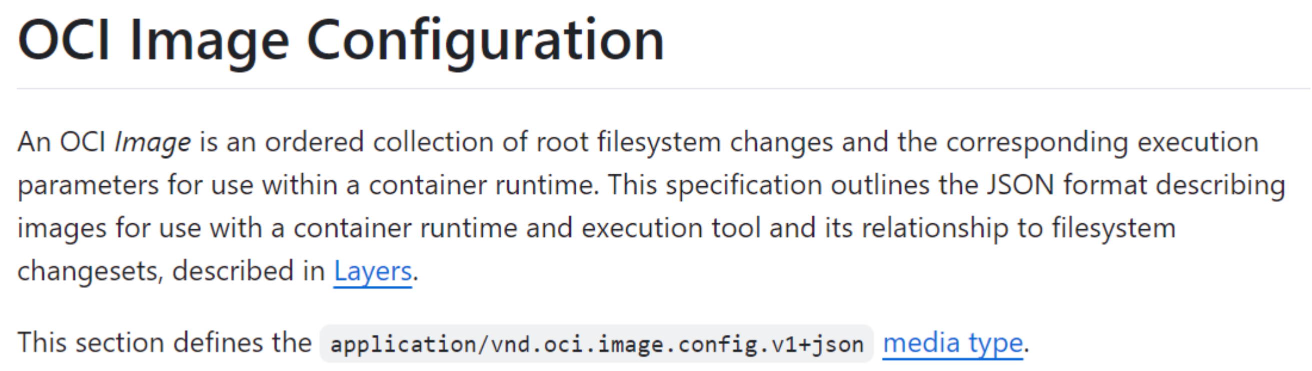
Claim 1	Accused Instrumentalities
	 **Portability** The flexibility of containers is based on its portability, ease of deployment, and smaller size compared to virtual machines. The Open Container Initiative (OCI), was formed to support fully interoperable container open standards with 3 specifications: • The Runtime Specification (runtime-spec) • The Image Specification (image-spec) • The Distribution Specification (distribution-spec) The OCI image specification defines an OCI Image, consisting of an image manifest, which contains metadata about contents and dependencies of the image; an image index (optional); a set of filesystem layers; and a configuration, such as arguments and environment variables. You can run the OCI compliant container image on any supported version of Linux or Windows, if you have the OCI compliant container runtime installed on the host. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/container-benefits.html  The open container initiative (OCI) images generated by docker build tools will continue to run in your Amazon EKS clusters as before. As an end-user of Kubernetes, you will not experience significant changes. You can continue to use Docker to build your containers outside the cluster. Visit this link to learn why Amazon EKS is discontinuing Dockershim support. For more information, see Kubernetes is Moving on From Dockershim: Commitments and Next Steps on the *Kubernetes Blog*. https://aws.amazon.com/blogs/containers/amazon-eks-now-supports-kubernetes-version-1-24/

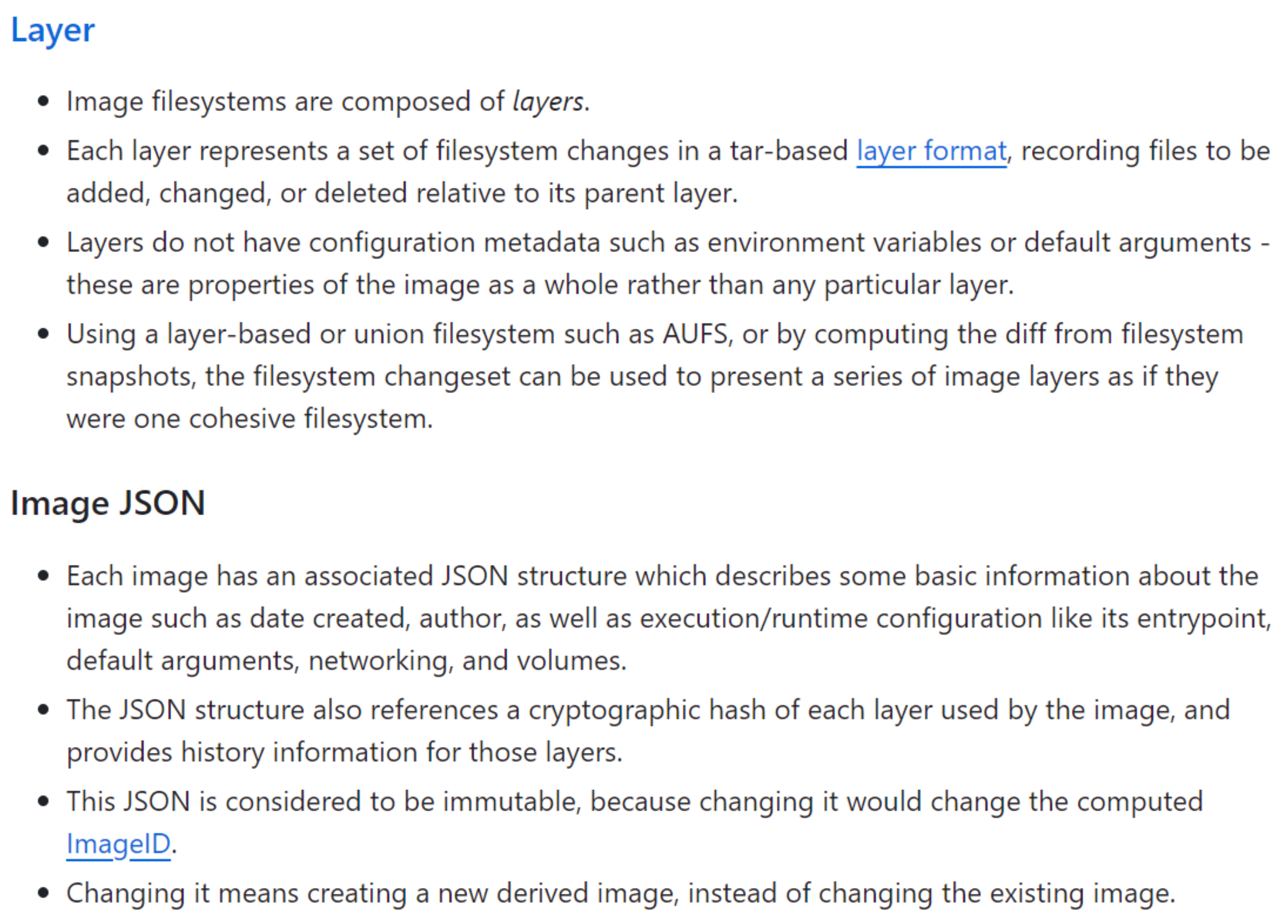
Claim 1	Accused Instrumentalities
	 A container runtime, also known as container engine, is a software component that can run containers on a host operating system. Container runtimes are responsible for loading container images from a repository, monitoring local system resources, isolating system resources for use of a container, and managing container lifecycle. They come in two forms: • High-level container runtimes (such as *containerd* and *CRI-O*) provide functions that run on top of low-level runtime. • Low-level runtimes are responsible for creating and running containers. The primary job of the low-level container runtimes is to provide container lifecycle management. These runtimes implement the Runtime Specification provided by the OCI(Open Container Initiative), a Linux Foundation project started by Docker, which aims to provide open standards for Linux containers. The default reference implementation for low level runtimes specified by OCI is *runc*. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/key-considerations.html

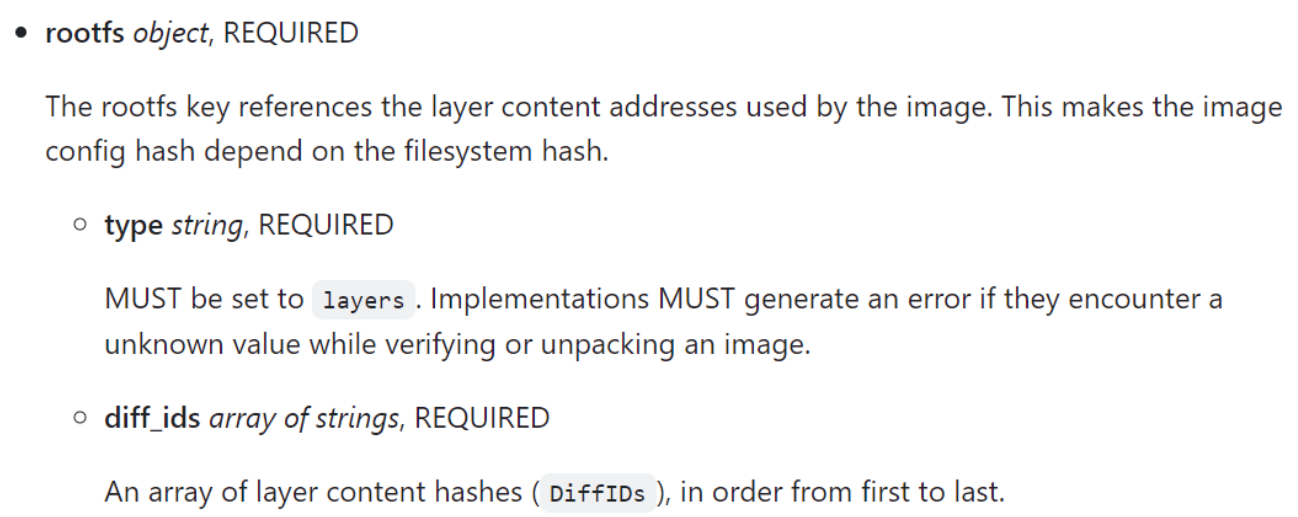
Claim 1	Accused Instrumentalities
	 Open Container Initiative Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

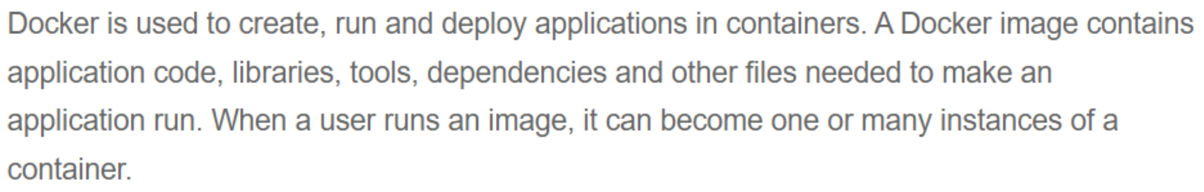
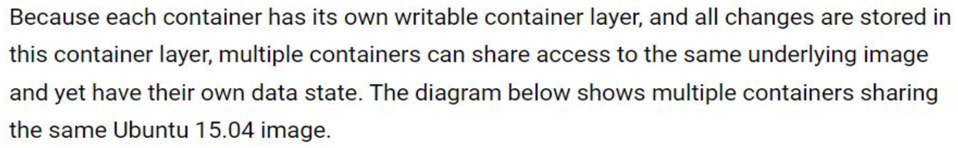
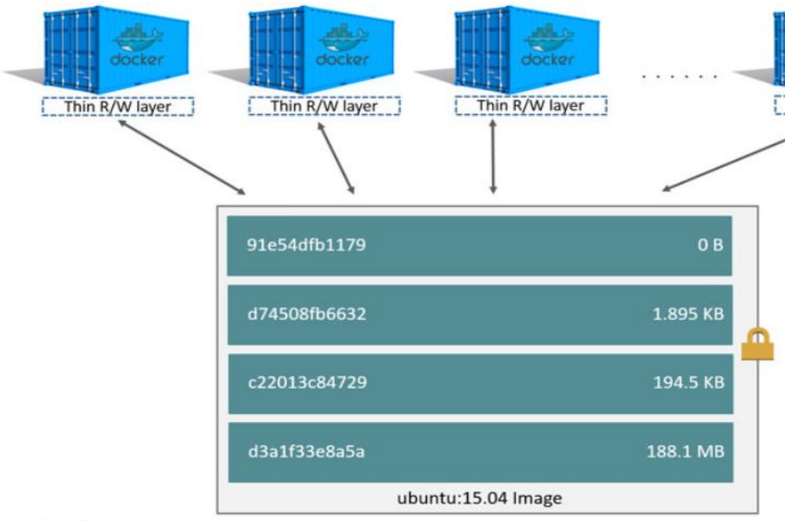
Claim 1	Accused Instrumentalities

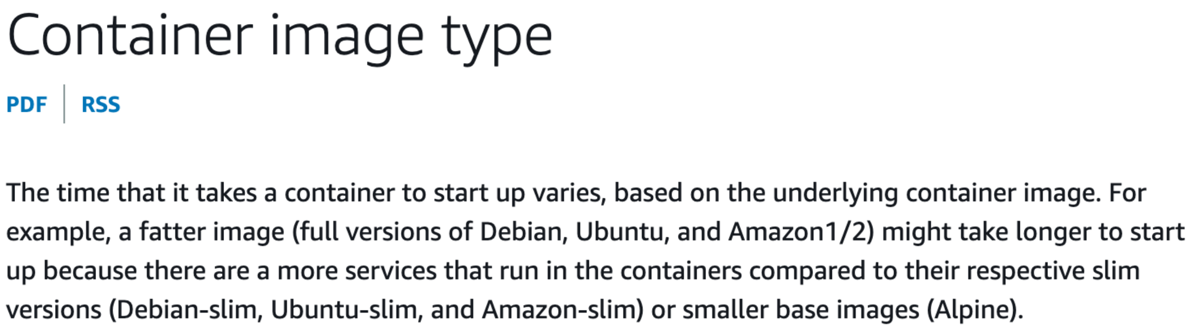
	
	Overview
	At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.
	`public class HelloWorld {`
	`  public static void main(String[] args) {`
	`    System.out.println("Hello, World");`
	`  }`
	`}`
	`/bin/java`
	`/opt/app.jar`
	`/lib/libc`
	`{ "manifests": { "platform": { "os": "linux", … }`
	`{ … "config": { "Cmd": [ "java", "-jar", "app.jar" ], … }`
	
	https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

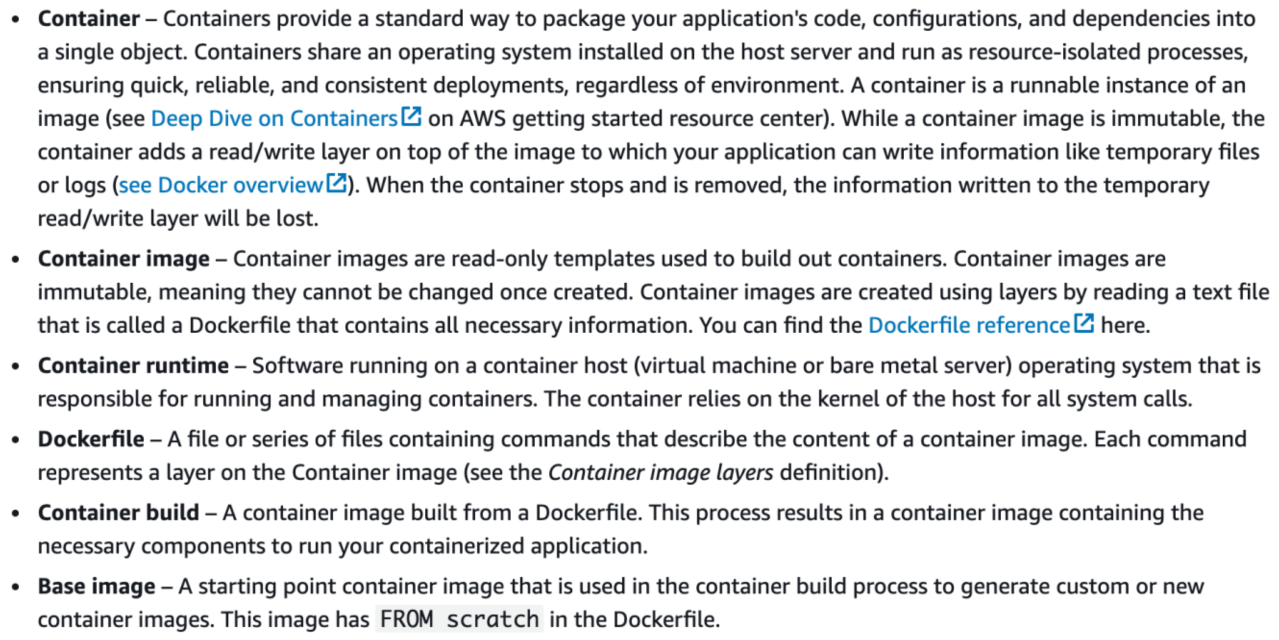
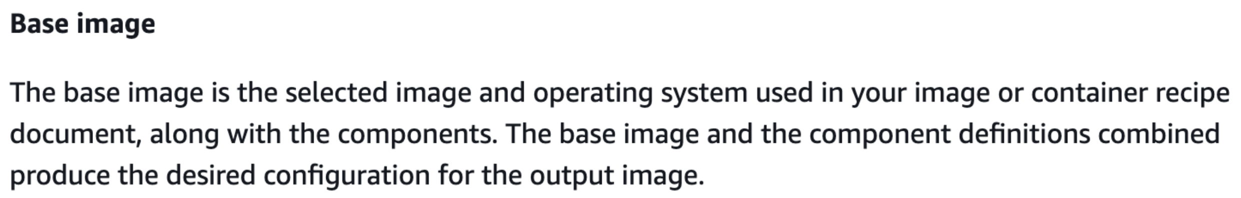
Claim 1	Accused Instrumentalities
	 **OCI Image Configuration** An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

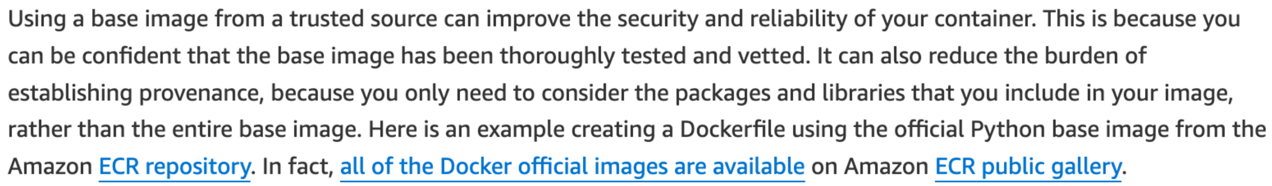
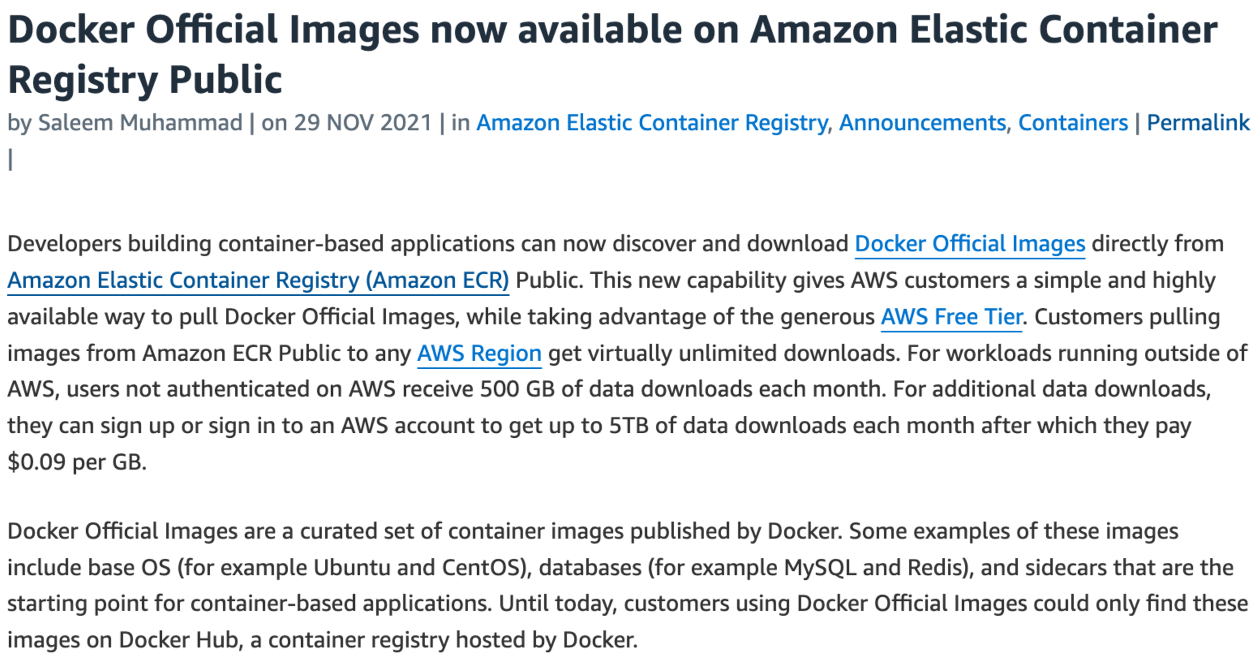
Claim 1	Accused Instrumentalities
	 **Layer** • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

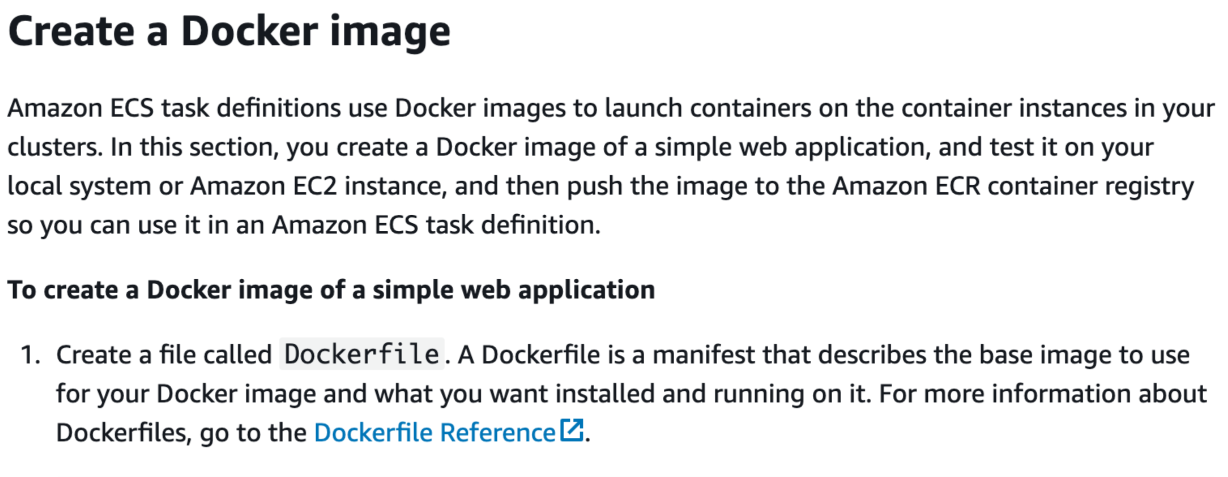
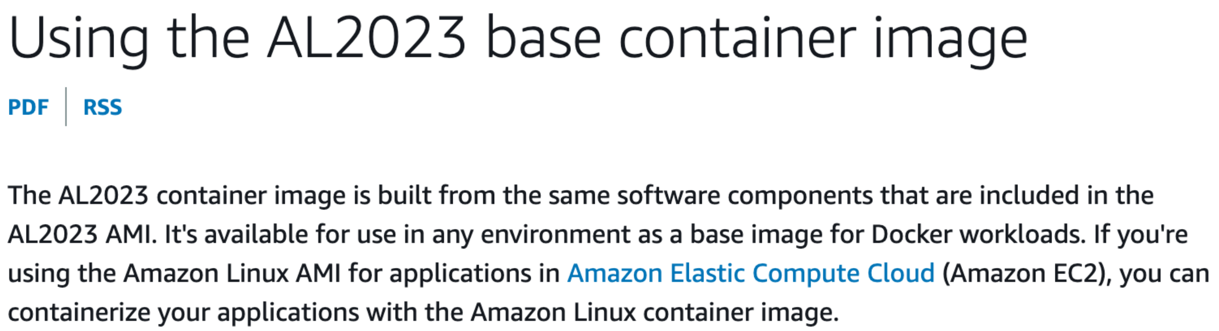
Claim 1	Accused Instrumentalities
	 • **rootfs** *object*, REQUIRED The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash. ○ **type** *string*, REQUIRED MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image. ○ **diff_ids** *array of strings*, REQUIRED An array of layer content hashes (`DiffIDs`), in order from first to last. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md
[1d] i) wherein some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications,	In each Accused Instrumentality, some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications. For example, a base image serves as a self-contained unit that encompasses all the necessary components for an application to run, including the application code, runtime environment, system tools, and dependencies (i.e., SLCSEs). The images are based on existing Linux distributions, such as Debian and Ubuntu, including essential system elements (i.e., functional replicas of OSCSEs). Each container image is based on a specific base image, which contains the application code, and dependencies, including system libraries or shared library critical system elements (SLCSEs). When the container runs the image, it creates a runtime instance of that container image. *See, e.g.*:

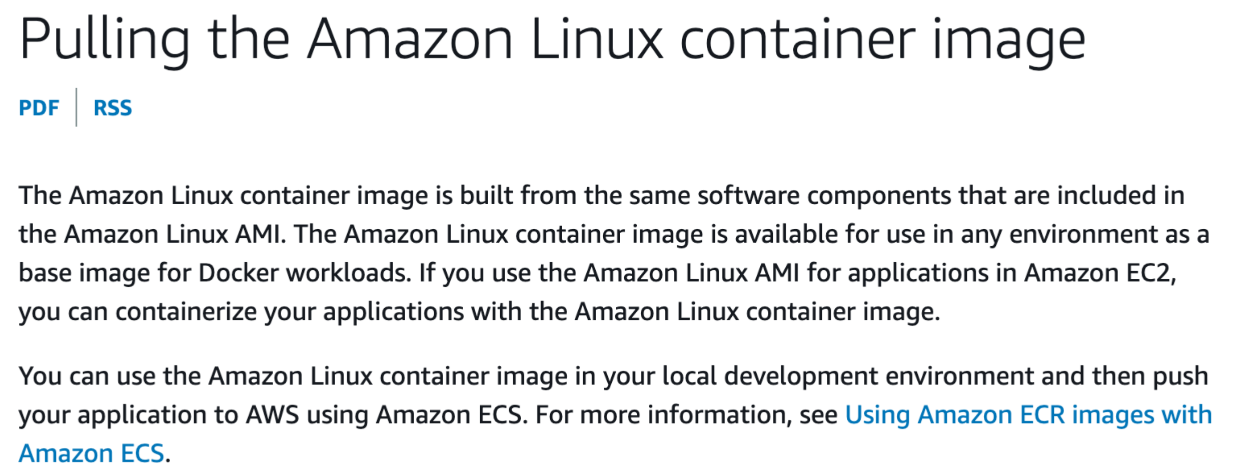
Claim 1	Accused Instrumentalities
	 Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. https://www.techtarget.com/searchitoperations/definition/Docker-image  Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  docker Thin R/W layer docker Thin R/W layer docker Thin R/W layer . . . . . . docker Thin R/W layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 Image https://docs.docker.com/storage/storagedriver/

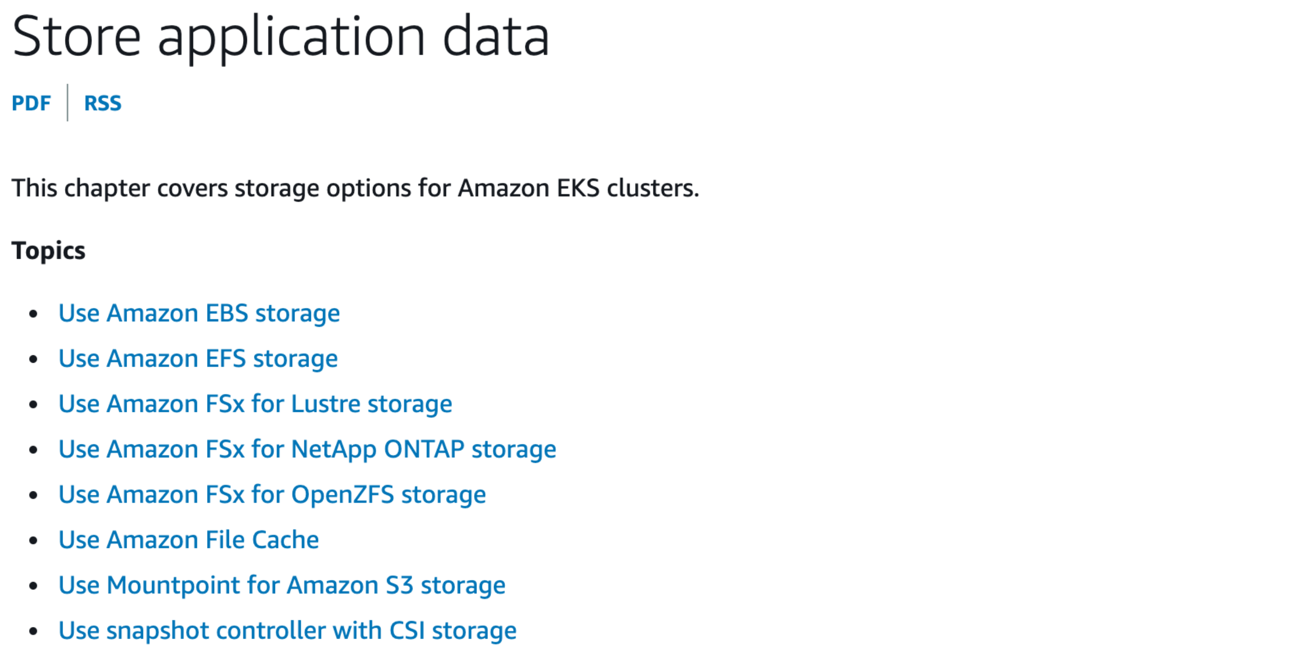
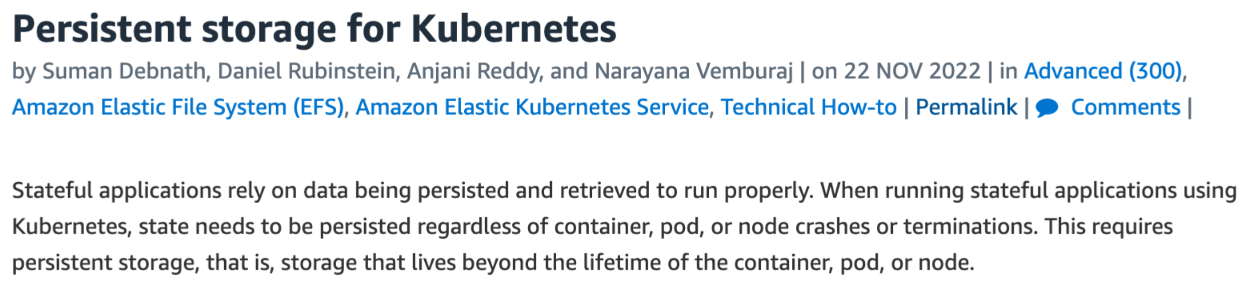
Claim 1	Accused Instrumentalities
	 Container image type PDF | RSS The time that it takes a container to start up varies, based on the underlying container image. For example, a fatter image (full versions of Debian, Ubuntu, and Amazon1/2) might take longer to start up because there are a more services that run in the containers compared to their respective slim versions (Debian-slim, Ubuntu-slim, and Amazon-slim) or smaller base images (Alpine). https://docs.aws.amazon.com/AmazonECS/latest/bestpracticesguide/container-type.html

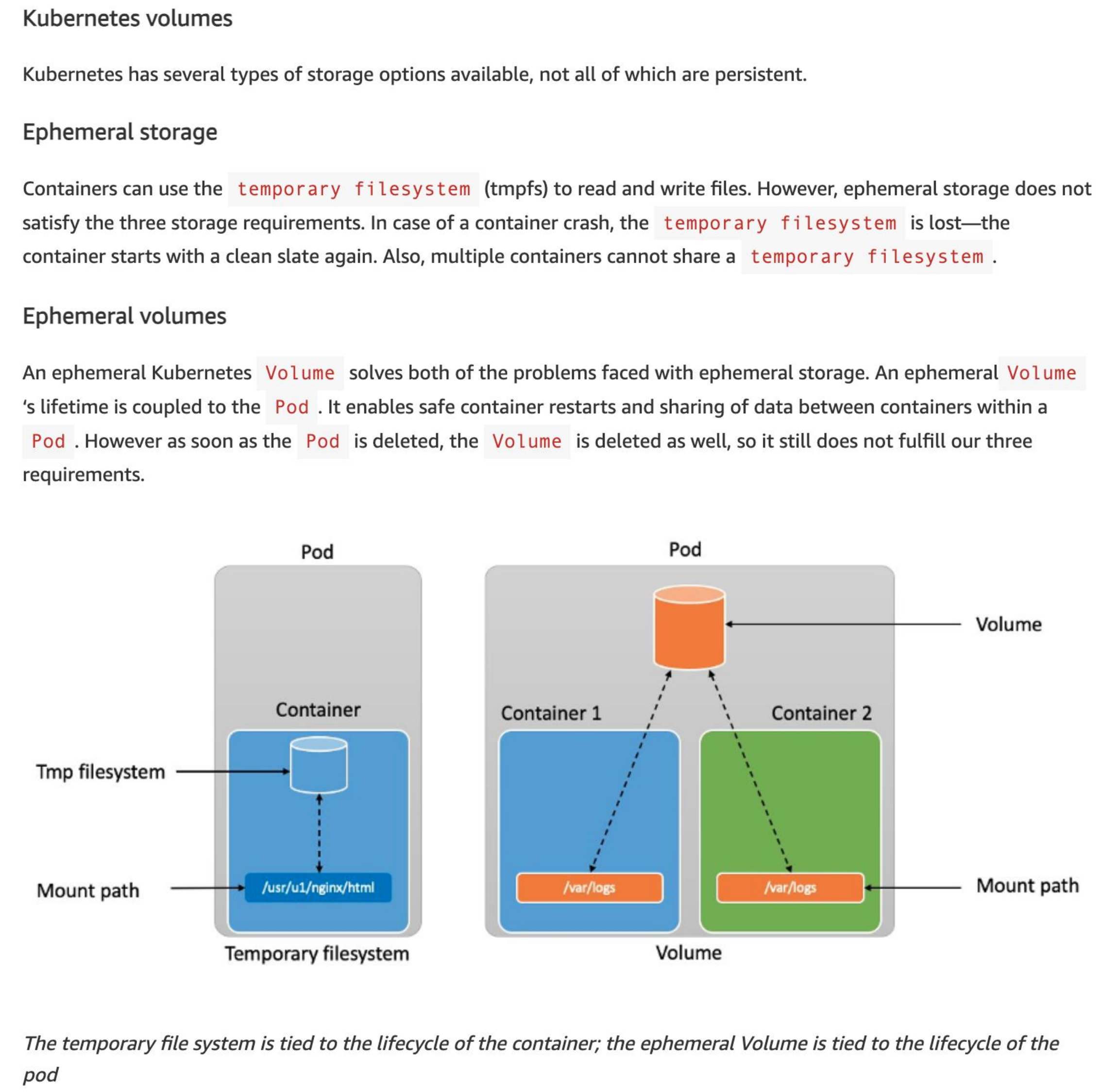
Claim 1	Accused Instrumentalities
	 • **Container** – Containers provide a standard way to package your application's code, configurations, and dependencies into a single object. Containers share an operating system installed on the host server and run as resource-isolated processes, ensuring quick, reliable, and consistent deployments, regardless of environment. A container is a runnable instance of an image (see Deep Dive on Containers on AWS getting started resource center). While a container image is immutable, the container adds a read/write layer on top of the image to which your application can write information like temporary files or logs (see Docker overview). When the container stops and is removed, the information written to the temporary read/write layer will be lost. • **Container image** – Container images are read-only templates used to build out containers. Container images are immutable, meaning they cannot be changed once created. Container images are created using layers by reading a text file that is called a Dockerfile that contains all necessary information. You can find the Dockerfile reference here. • **Container runtime** – Software running on a container host (virtual machine or bare metal server) operating system that is responsible for running and managing containers. The container relies on the kernel of the host for all system calls. • **Dockerfile** – A file or series of files containing commands that describe the content of a container image. Each command represents a layer on the Container image (see the *Container image layers* definition). • **Container build** – A container image built from a Dockerfile. This process results in a container image containing the necessary components to run your containerized application. • **Base image** – A starting point container image that is used in the container build process to generate custom or new container images. This image has `FROM scratch` in the Dockerfile. https://docs.aws.amazon.com/wellarchitected/latest/container-build-lens/container-technology-terminology.html  **Base image** The base image is the selected image and operating system used in your image or container recipe document, along with the components. The base image and the component definitions combined produce the desired configuration for the output image. https://docs.aws.amazon.com/imagebuilder/latest/userguide/what-is-image-builder.html

Claim 1	Accused Instrumentalities
	 Using a base image from a trusted source can improve the security and reliability of your container. This is because you can be confident that the base image has been thoroughly tested and vetted. It can also reduce the burden of establishing provenance, because you only need to consider the packages and libraries that you include in your image, rather than the entire base image. Here is an example creating a Dockerfile using the official Python base image from the Amazon ECR repository. In fact, all of the Docker official images are available on Amazon ECR public gallery. https://aws.amazon.com/blogs/containers/building-better-container-images/  **Docker Official Images now available on Amazon Elastic Container Registry Public** by Saleem Muhammad | on 29 NOV 2021 | in Amazon Elastic Container Registry, Announcements, Containers | Permalink | Developers building container-based applications can now discover and download Docker Official Images directly from Amazon Elastic Container Registry (Amazon ECR) Public. This new capability gives AWS customers a simple and highly available way to pull Docker Official Images, while taking advantage of the generous AWS Free Tier. Customers pulling images from Amazon ECR Public to any AWS Region get virtually unlimited downloads. For workloads running outside of AWS, users not authenticated on AWS receive 500 GB of data downloads each month. For additional data downloads, they can sign up or sign in to an AWS account to get up to 5TB of data downloads each month after which they pay $0.09 per GB. Docker Official Images are a curated set of container images published by Docker. Some examples of these images include base OS (for example Ubuntu and CentOS), databases (for example MySQL and Redis), and sidecars that are the starting point for container-based applications. Until today, customers using Docker Official Images could only find these images on Docker Hub, a container registry hosted by Docker. https://aws.amazon.com/blogs/containers/docker-official-images-now-available-on-amazon-elastic-container-registry-public/

Claim 1	Accused Instrumentalities
	 **Create a Docker image** Amazon ECS task definitions use Docker images to launch containers on the container instances in your clusters. In this section, you create a Docker image of a simple web application, and test it on your local system or Amazon EC2 instance, and then push the image to the Amazon ECR container registry so you can use it in an Amazon ECS task definition. **To create a Docker image of a simple web application** 1. Create a file called `Dockerfile`. A Dockerfile is a manifest that describes the base image to use for your Docker image and what you want installed and running on it. For more information about Dockerfiles, go to the Dockerfile Reference. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create-container-image.html  Using the AL2023 base container image PDF | RSS The AL2023 container image is built from the same software components that are included in the AL2023 AMI. It's available for use in any environment as a base image for Docker workloads. If you're using the Amazon Linux AMI for applications in Amazon Elastic Compute Cloud (Amazon EC2), you can containerize your applications with the Amazon Linux container image. https://docs.aws.amazon.com/linux/al2023/ug/base-container.html

Claim 1	Accused Instrumentalities
	 Pulling the Amazon Linux container image PDF | RSS The Amazon Linux container image is built from the same software components that are included in the Amazon Linux AMI. The Amazon Linux container image is available for use in any environment as a base image for Docker workloads. If you use the Amazon Linux AMI for applications in Amazon EC2, you can containerize your applications with the Amazon Linux container image. You can use the Amazon Linux container image in your local development environment and then push your application to AWS using Amazon ECS. For more information, see Using Amazon ECR images with Amazon ECS. https://docs.aws.amazon.com/AmazonECR/latest/userguide/amazon_linux_container_image.html

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
Store application data
PDF | RSS
This chapter covers storage options for Amazon EKS clusters.
Topics
• Use Amazon EBS storage
• Use Amazon EFS storage
• Use Amazon FSx for Lustre storage
• Use Amazon FSx for NetApp ONTAP storage
• Use Amazon FSx for OpenZFS storage
• Use Amazon File Cache
• Use Mountpoint for Amazon S3 storage
• Use snapshot controller with CSI storage
https://docs.aws.amazon.com/eks/latest/userguide/storage.html

Persistent storage for Kubernetes
by Suman Debnath, Daniel Rubinstein, Anjani Reddy, and Narayana Vemburaj | on 22 NOV 2022 | in Advanced (300), Amazon Elastic File System (EFS), Amazon Elastic Kubernetes Service, Technical How-to | Permalink | 💬 Comments |
Stateful applications rely on data being persisted and retrieved to run properly. When running stateful applications using Kubernetes, state needs to be persisted regardless of container, pod, or node crashes or terminations. This requires persistent storage, that is, storage that lives beyond the lifetime of the container, pod, or node.
https://aws.amazon.com/blogs/storage/persistent-storage-for-kubernetes/</td></tr>
</table>



Kubernetes volumes

Kubernetes has several types of storage options available, not all of which are persistent.

Ephemeral storage

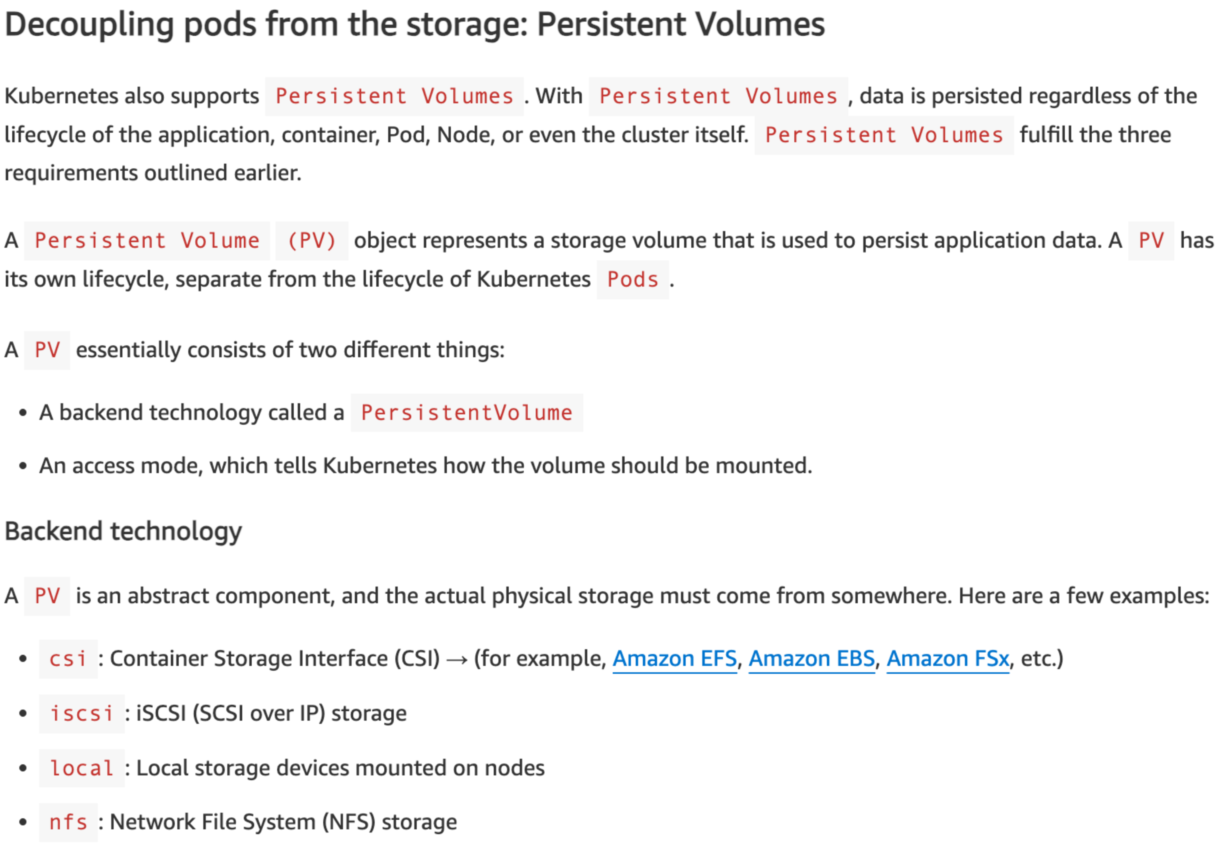
Containers can use the `temporary filesystem` (tmpfs) to read and write files. However, ephemeral storage does not satisfy the three storage requirements. In case of a container crash, the `temporary filesystem` is lost—the container starts with a clean slate again. Also, multiple containers cannot share a `temporary filesystem` .

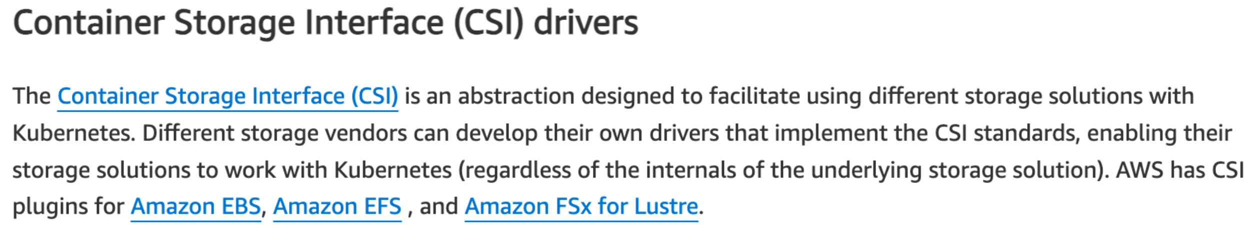
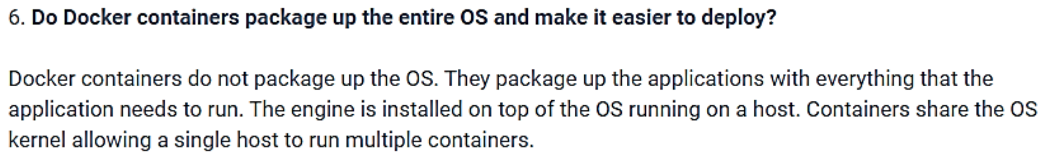
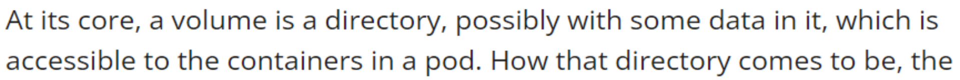
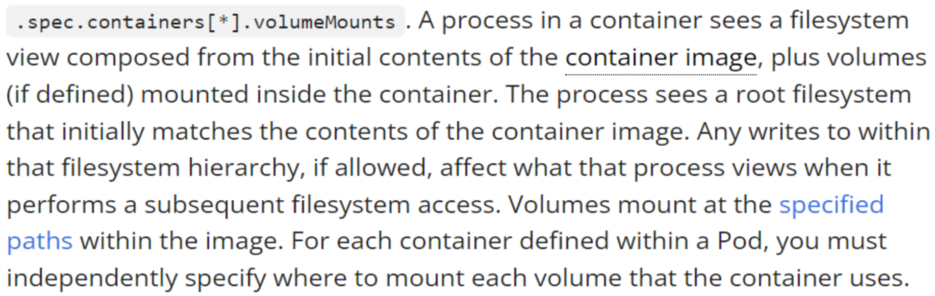
Ephemeral volumes

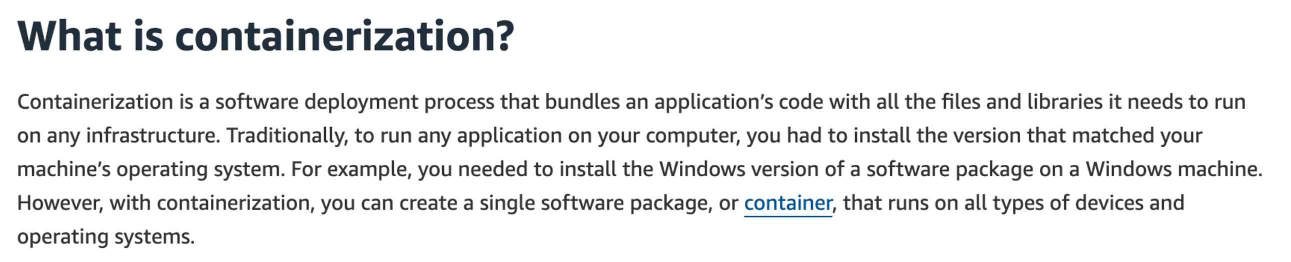
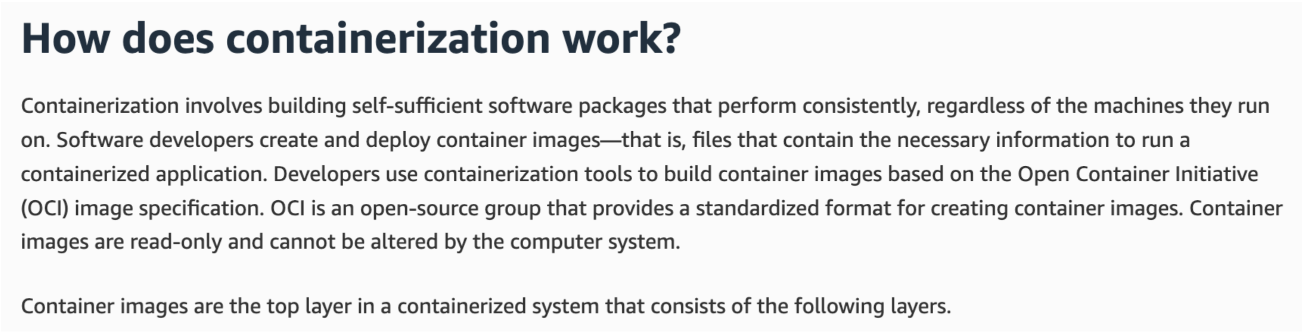
An ephemeral Kubernetes `Volume` solves both of the problems faced with ephemeral storage. An ephemeral `Volume` 's lifetime is coupled to the `Pod` . It enables safe container restarts and sharing of data between containers within a `Pod` . However as soon as the `Pod` is deleted, the `Volume` is deleted as well, so it still does not fulfill our three requirements.

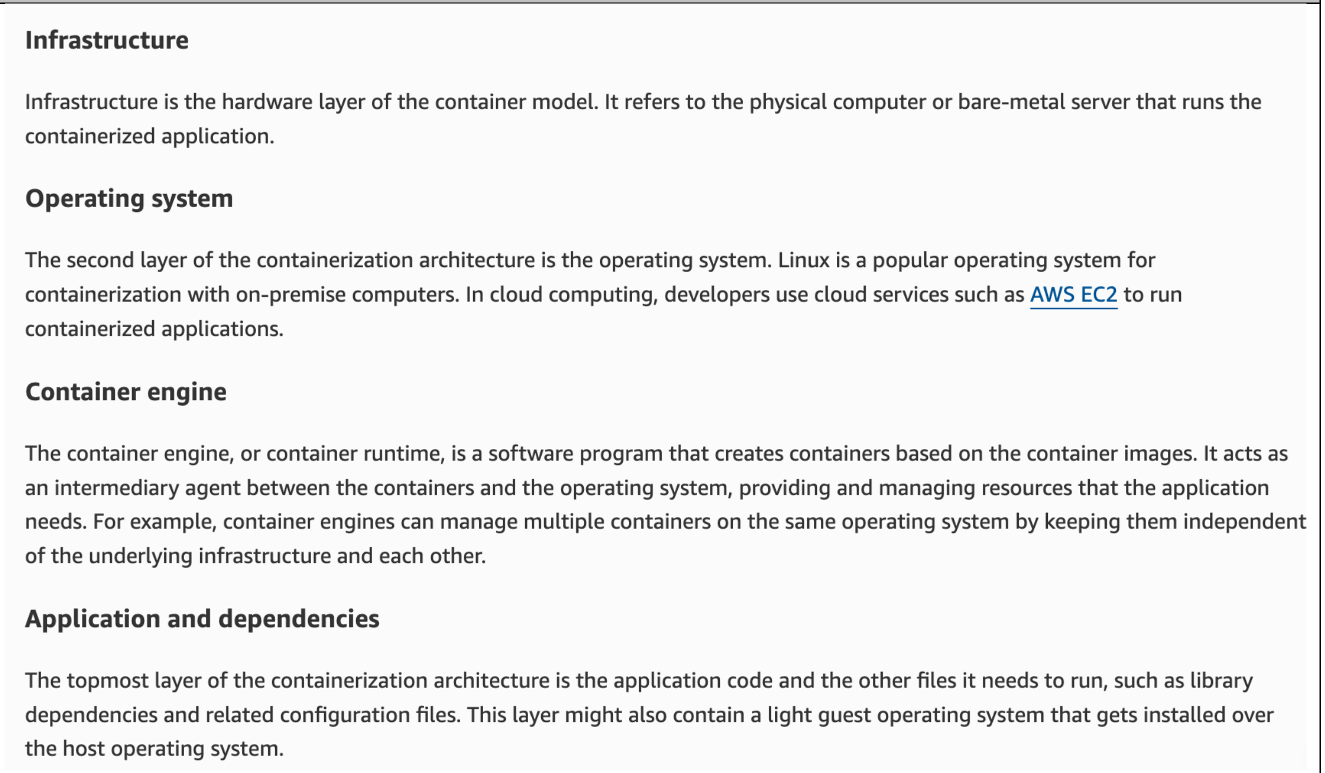
The temporary file system is tied to the lifecycle of the container; the ephemeral Volume is tied to the lifecycle of the pod

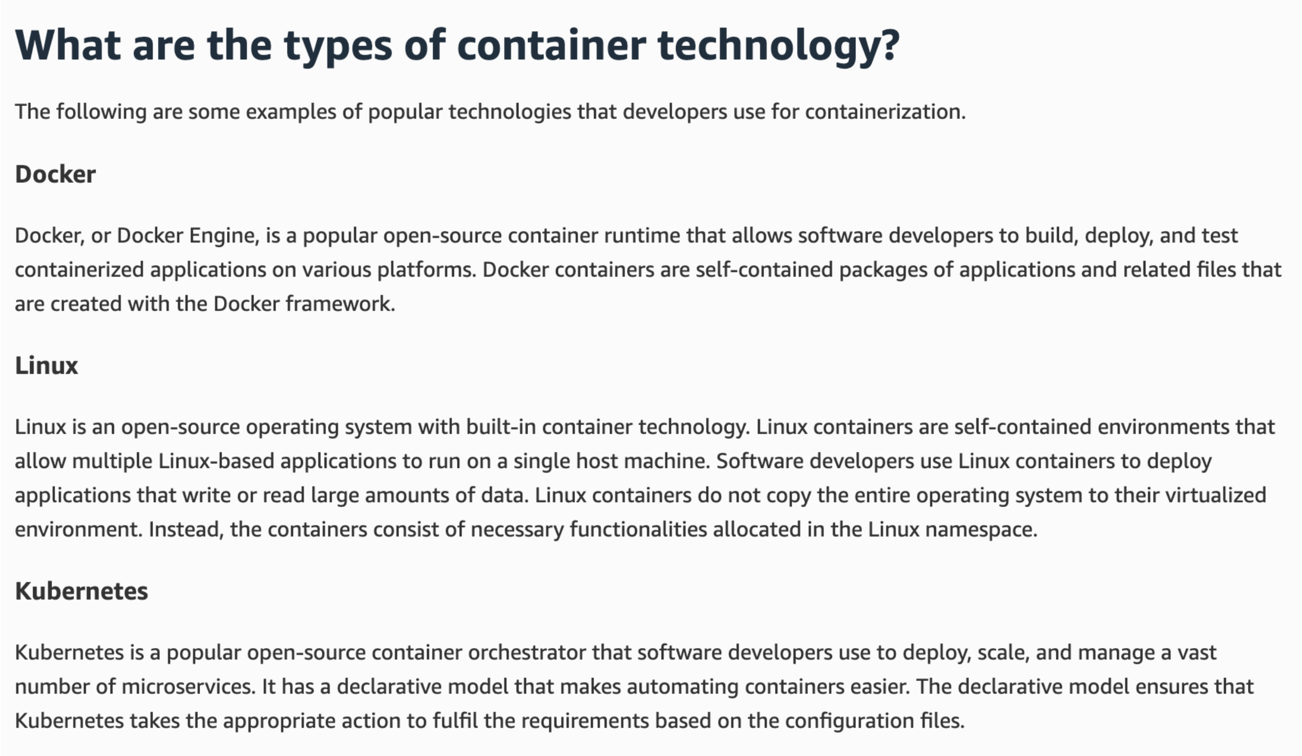
https://aws.amazon.com/blogs/storage/persistent-storage-for-kubernetes/

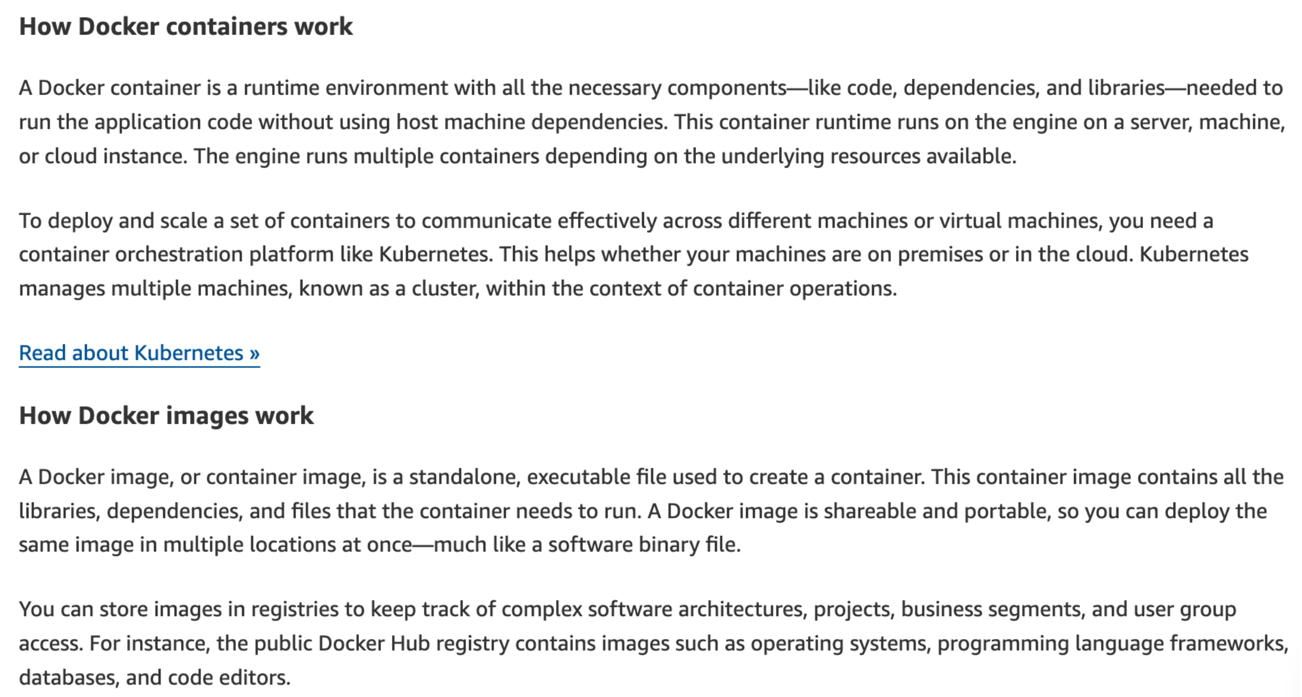
Claim 1	Accused Instrumentalities
	 **Decoupling pods from the storage: Persistent Volumes** Kubernetes also supports `Persistent Volumes`. With `Persistent Volumes`, data is persisted regardless of the lifecycle of the application, container, Pod, Node, or even the cluster itself. `Persistent Volumes` fulfill the three requirements outlined earlier. A `Persistent Volume` `(PV)` object represents a storage volume that is used to persist application data. A `PV` has its own lifecycle, separate from the lifecycle of Kubernetes `Pods`. A `PV` essentially consists of two different things: - A backend technology called a `PersistentVolume` - An access mode, which tells Kubernetes how the volume should be mounted. **Backend technology** A `PV` is an abstract component, and the actual physical storage must come from somewhere. Here are a few examples: - `csi`: Container Storage Interface (CSI) → (for example, Amazon EFS, Amazon EBS, Amazon FSx, etc.) - `iscsi`: iSCSI (SCSI over IP) storage - `local`: Local storage devices mounted on nodes - `nfs`: Network File System (NFS) storage https://aws.amazon.com/blogs/storage/persistent-storage-for-kubernetes/

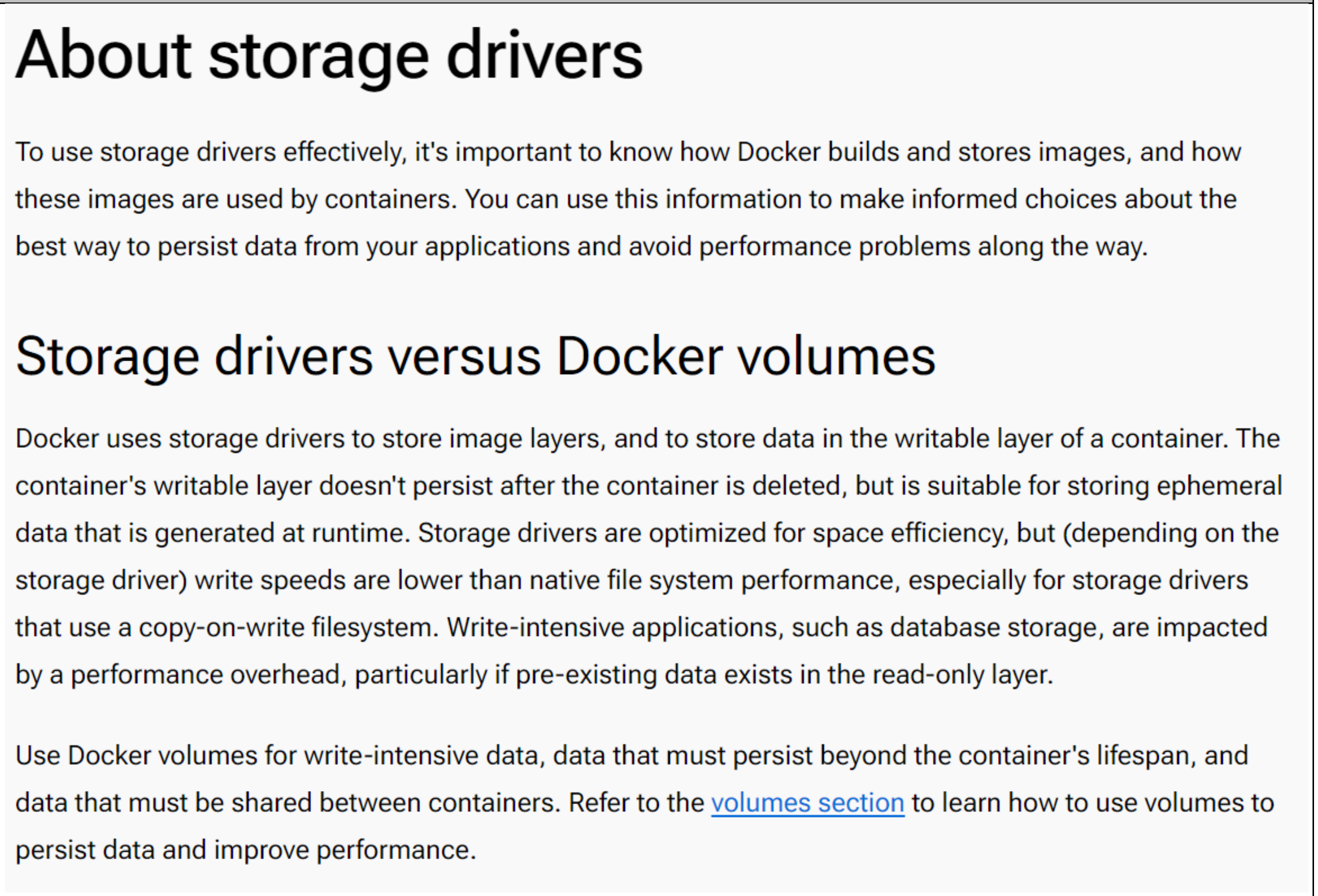
Claim 1	Accused Instrumentalities
	 **Container Storage Interface (CSI) drivers** The Container Storage Interface (CSI) is an abstraction designed to facilitate using different storage solutions with Kubernetes. Different storage vendors can develop their own drivers that implement the CSI standards, enabling their storage solutions to work with Kubernetes (regardless of the internals of the underlying storage solution). AWS has CSI plugins for Amazon EBS, Amazon EFS , and Amazon FSx for Lustre. https://aws.amazon.com/blogs/storage/persistent-storage-for-kubernetes/  6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/  At its core, a volume is a directory, possibly with some data in it, which is accessible to the containers in a pod. How that directory comes to be, the  `.spec.containers[*].volumeMounts`. A process in a container sees a filesystem view composed from the initial contents of the container image, plus volumes (if defined) mounted inside the container. The process sees a root filesystem that initially matches the contents of the container image. Any writes to within that filesystem hierarchy, if allowed, affect what that process views when it performs a subsequent filesystem access. Volumes mount at the specified paths within the image. For each container defined within a Pod, you must independently specify where to mount each volume that the container uses. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	 **What is containerization?** Containerization is a software deployment process that bundles an application's code with all the files and libraries it needs to run on any infrastructure. Traditionally, to run any application on your computer, you had to install the version that matched your machine's operating system. For example, you needed to install the Windows version of a software package on a Windows machine. However, with containerization, you can create a single software package, or container, that runs on all types of devices and operating systems. https://aws.amazon.com/what-is/containerization/  **How does containerization work?** Containerization involves building self-sufficient software packages that perform consistently, regardless of the machines they run on. Software developers create and deploy container images—that is, files that contain the necessary information to run a containerized application. Developers use containerization tools to build container images based on the Open Container Initiative (OCI) image specification. OCI is an open-source group that provides a standardized format for creating container images. Container images are read-only and cannot be altered by the computer system. Container images are the top layer in a containerized system that consists of the following layers. https://aws.amazon.com/what-is/containerization/

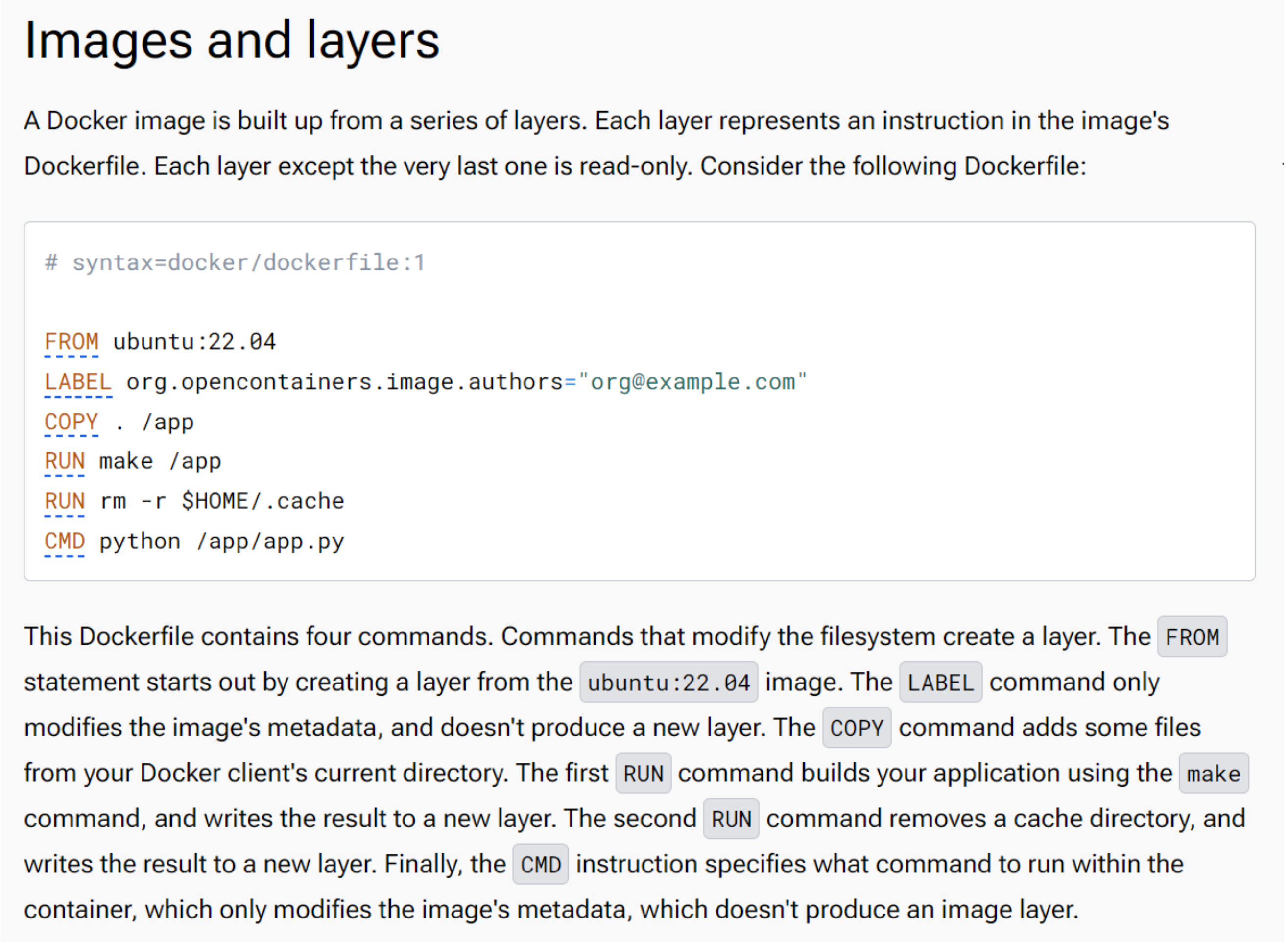
Claim 1	Accused Instrumentalities
	 **Infrastructure** Infrastructure is the hardware layer of the container model. It refers to the physical computer or bare-metal server that runs the containerized application. **Operating system** The second layer of the containerization architecture is the operating system. Linux is a popular operating system for containerization with on-premise computers. In cloud computing, developers use cloud services such as AWS EC2 to run containerized applications. **Container engine** The container engine, or container runtime, is a software program that creates containers based on the container images. It acts as an intermediary agent between the containers and the operating system, providing and managing resources that the application needs. For example, container engines can manage multiple containers on the same operating system by keeping them independent of the underlying infrastructure and each other. **Application and dependencies** The topmost layer of the containerization architecture is the application code and the other files it needs to run, such as library dependencies and related configuration files. This layer might also contain a light guest operating system that gets installed over the host operating system. https://aws.amazon.com/what-is/containerization/

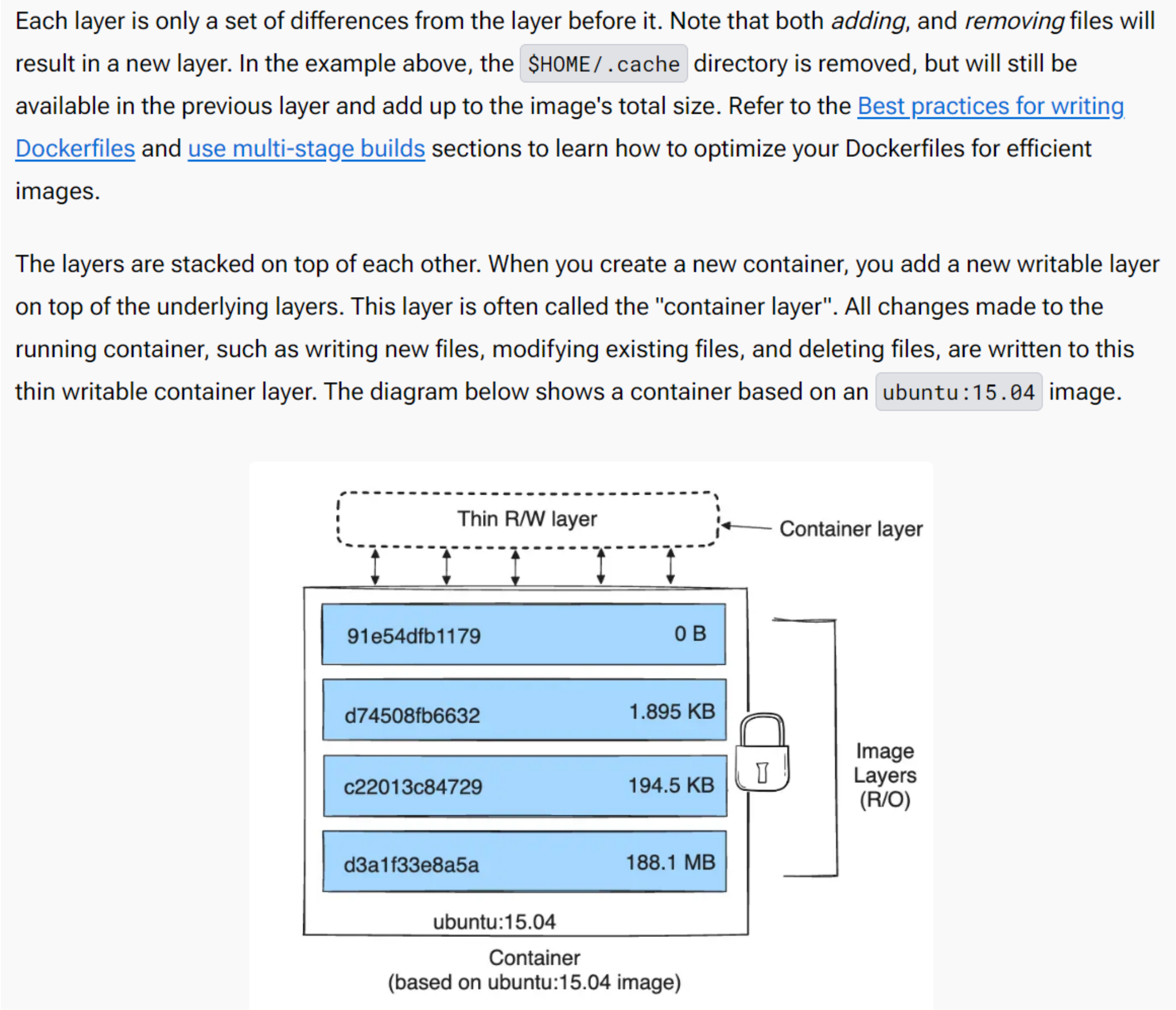
Claim 1	Accused Instrumentalities
	 **What are the types of container technology?** The following are some examples of popular technologies that developers use for containerization. **Docker** Docker, or Docker Engine, is a popular open-source container runtime that allows software developers to build, deploy, and test containerized applications on various platforms. Docker containers are self-contained packages of applications and related files that are created with the Docker framework. **Linux** Linux is an open-source operating system with built-in container technology. Linux containers are self-contained environments that allow multiple Linux-based applications to run on a single host machine. Software developers use Linux containers to deploy applications that write or read large amounts of data. Linux containers do not copy the entire operating system to their virtualized environment. Instead, the containers consist of necessary functionalities allocated in the Linux namespace. **Kubernetes** Kubernetes is a popular open-source container orchestrator that software developers use to deploy, scale, and manage a vast number of microservices. It has a declarative model that makes automating containers easier. The declarative model ensures that Kubernetes takes the appropriate action to fulfil the requirements based on the configuration files. https://aws.amazon.com/what-is/containerization/

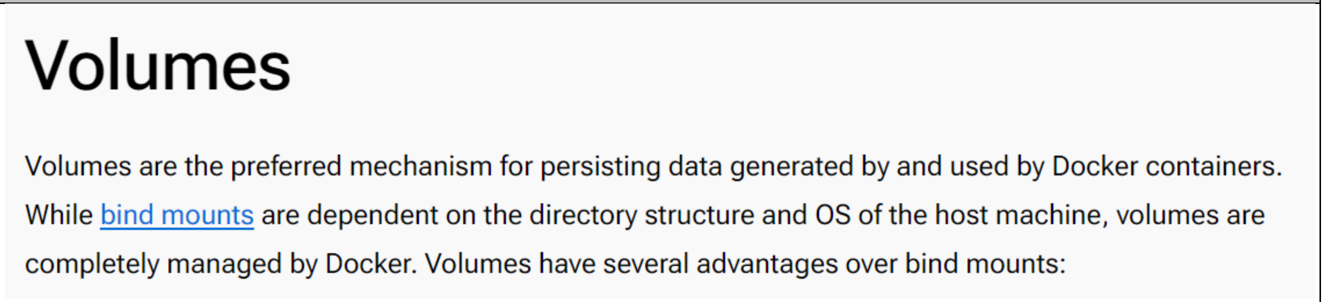
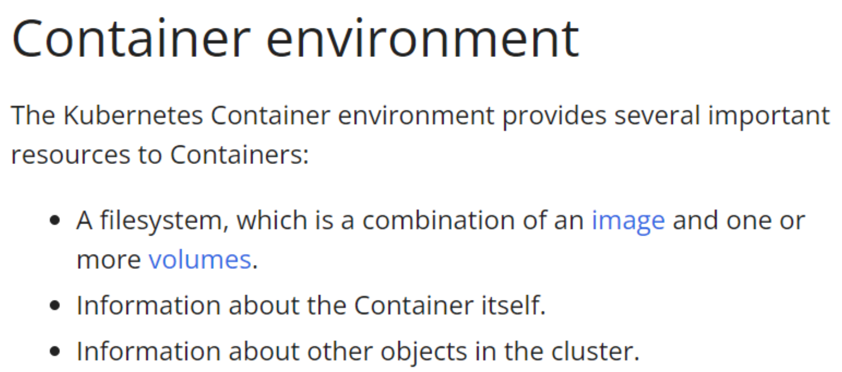
Claim 1	Accused Instrumentalities
	 **How Docker containers work** A Docker container is a runtime environment with all the necessary components—like code, dependencies, and libraries—needed to run the application code without using host machine dependencies. This container runtime runs on the engine on a server, machine, or cloud instance. The engine runs multiple containers depending on the underlying resources available. To deploy and scale a set of containers to communicate effectively across different machines or virtual machines, you need a container orchestration platform like Kubernetes. This helps whether your machines are on premises or in the cloud. Kubernetes manages multiple machines, known as a cluster, within the context of container operations. Read about Kubernetes » **How Docker images work** A Docker image, or container image, is a standalone, executable file used to create a container. This container image contains all the libraries, dependencies, and files that the container needs to run. A Docker image is shareable and portable, so you can deploy the same image in multiple locations at once—much like a software binary file. You can store images in registries to keep track of complex software architectures, projects, business segments, and user group access. For instance, the public Docker Hub registry contains images such as operating systems, programming language frameworks, databases, and code editors. https://aws.amazon.com/compare/the-difference-between-docker-images-and-containers/

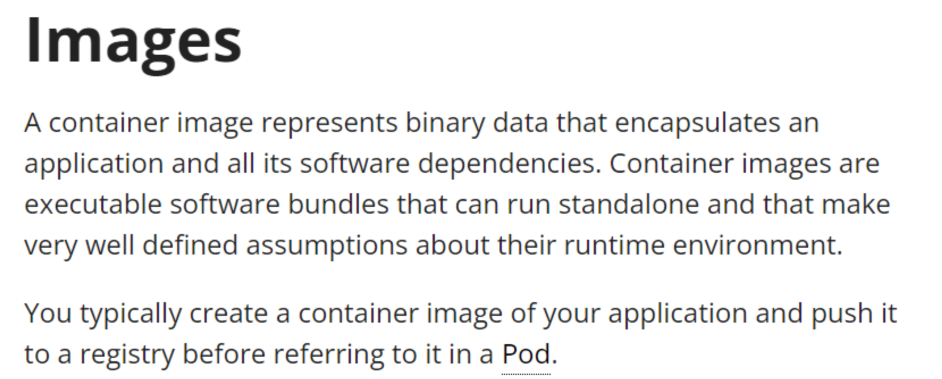
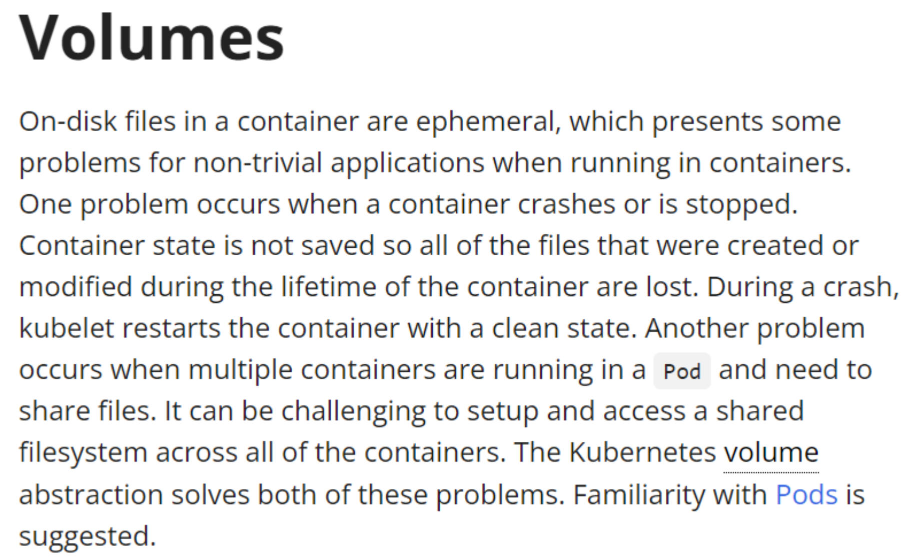
Claim 1	Accused Instrumentalities
	 About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

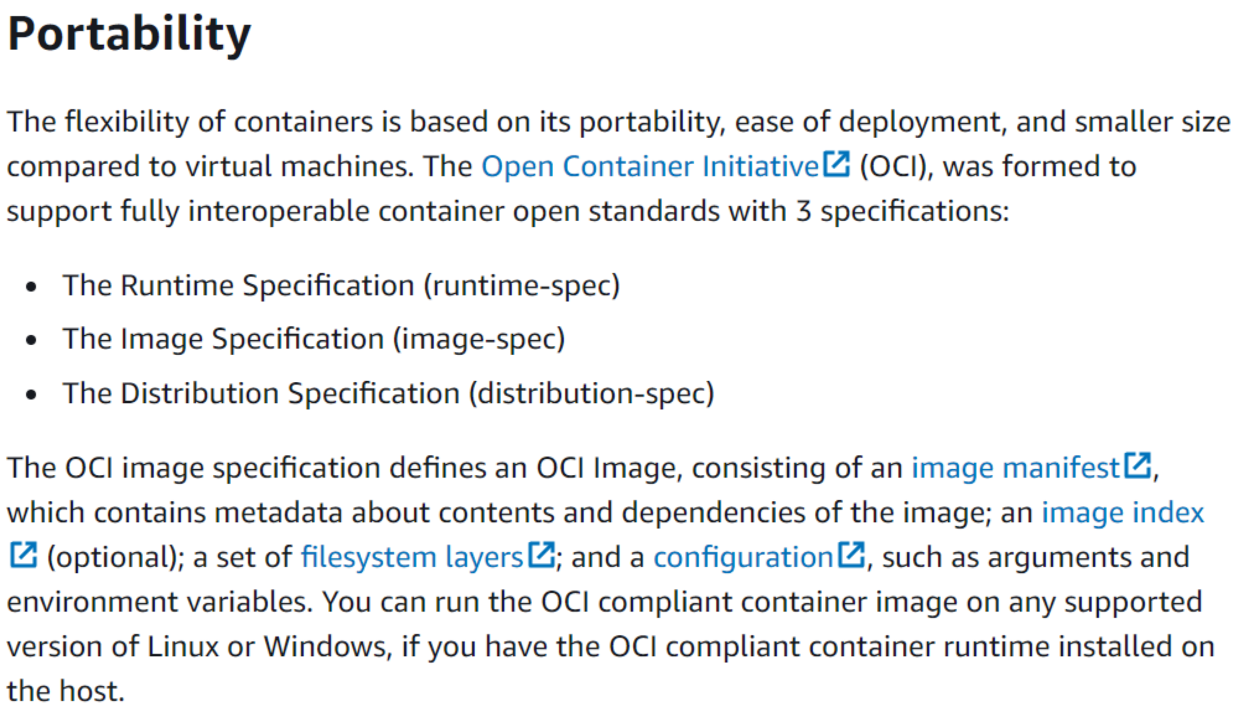
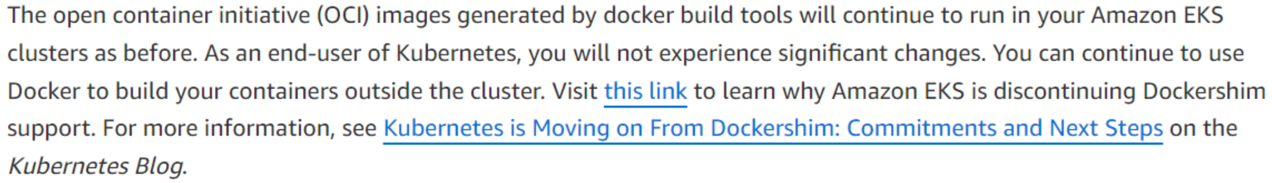
Claim 1	Accused Instrumentalities

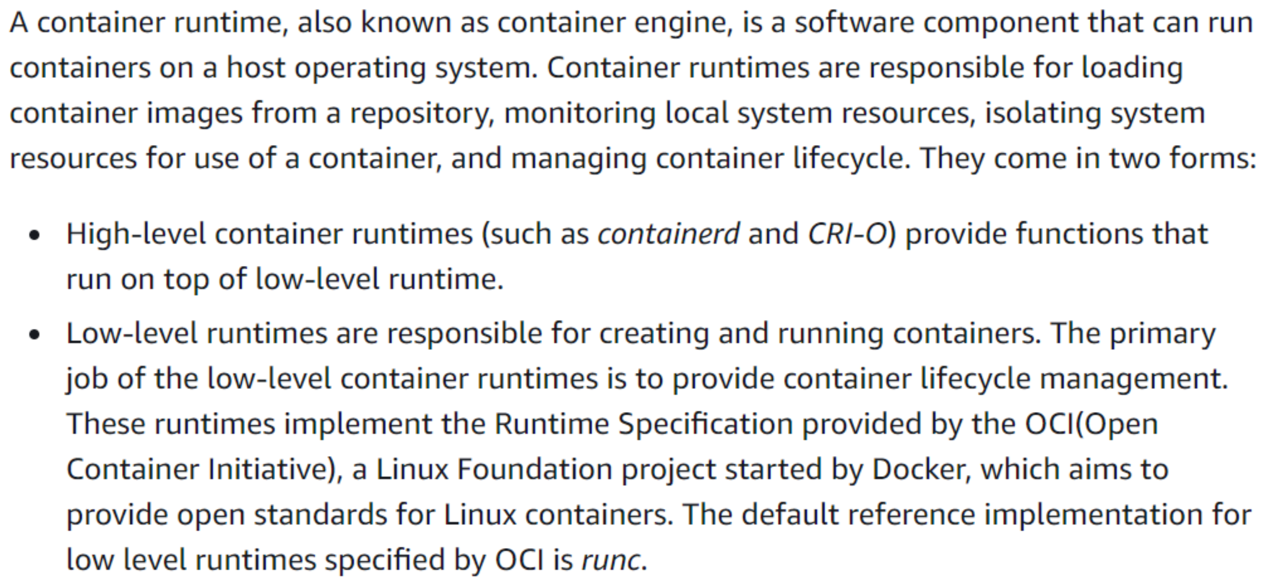
	
	Images and layers
	A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:
	`# syntax=docker/dockerfile:1`
	
	`FROM ubuntu:22.04`
	`LABEL org.opencontainers.image.authors="org@example.com"`
	`COPY . /app`
	`RUN make /app`
	`RUN rm -r $HOME/.cache`
	`CMD python /app/app.py`
	This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.
	https://docs.docker.com/storage/storagedriver/

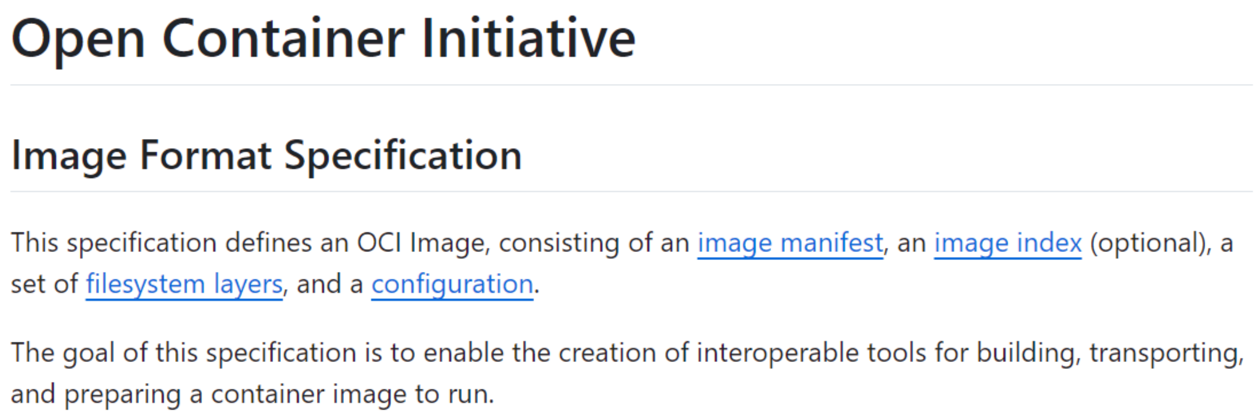
	 Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image. Thin R/W layer Container layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB Image Layers (R/O) ubuntu:15.04 Container (based on ubuntu:15.04 image) https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	 Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/  Container environment The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

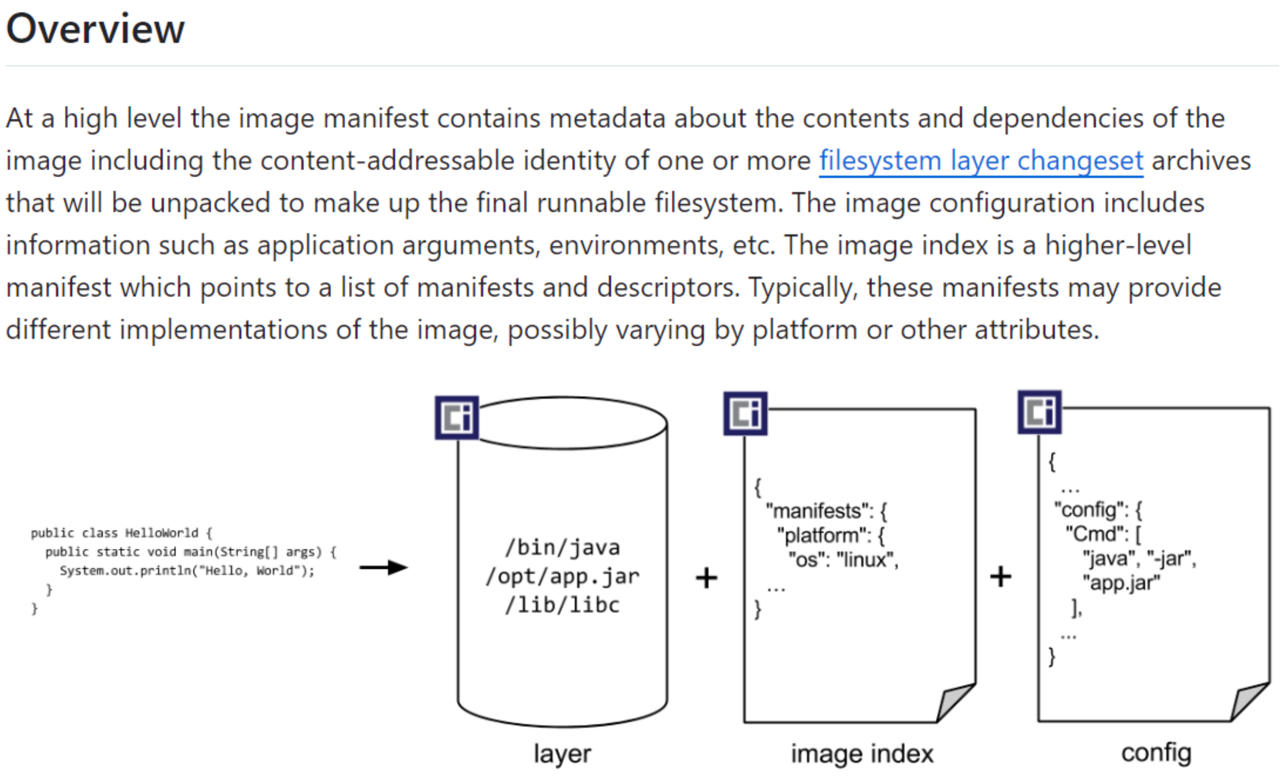
Claim 1	Accused Instrumentalities
	 **Images** A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/  **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

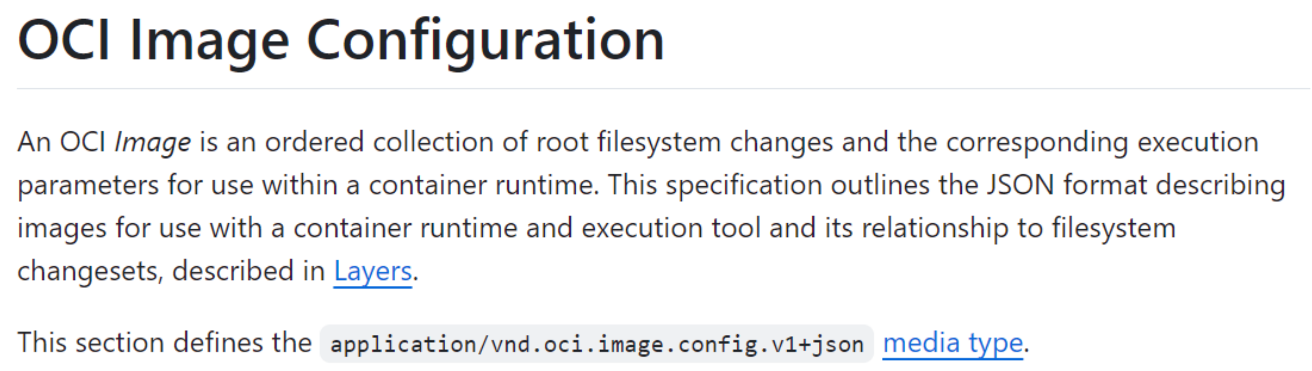
Claim 1	Accused Instrumentalities
	 **Portability** The flexibility of containers is based on its portability, ease of deployment, and smaller size compared to virtual machines. The Open Container Initiative (OCI), was formed to support fully interoperable container open standards with 3 specifications: • The Runtime Specification (runtime-spec) • The Image Specification (image-spec) • The Distribution Specification (distribution-spec) The OCI image specification defines an OCI Image, consisting of an image manifest, which contains metadata about contents and dependencies of the image; an image index (optional); a set of filesystem layers; and a configuration, such as arguments and environment variables. You can run the OCI compliant container image on any supported version of Linux or Windows, if you have the OCI compliant container runtime installed on the host. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/container-benefits.html  The open container initiative (OCI) images generated by docker build tools will continue to run in your Amazon EKS clusters as before. As an end-user of Kubernetes, you will not experience significant changes. You can continue to use Docker to build your containers outside the cluster. Visit this link to learn why Amazon EKS is discontinuing Dockershim support. For more information, see Kubernetes is Moving on From Dockershim: Commitments and Next Steps on the *Kubernetes Blog*. https://aws.amazon.com/blogs/containers/amazon-eks-now-supports-kubernetes-version-1-24/

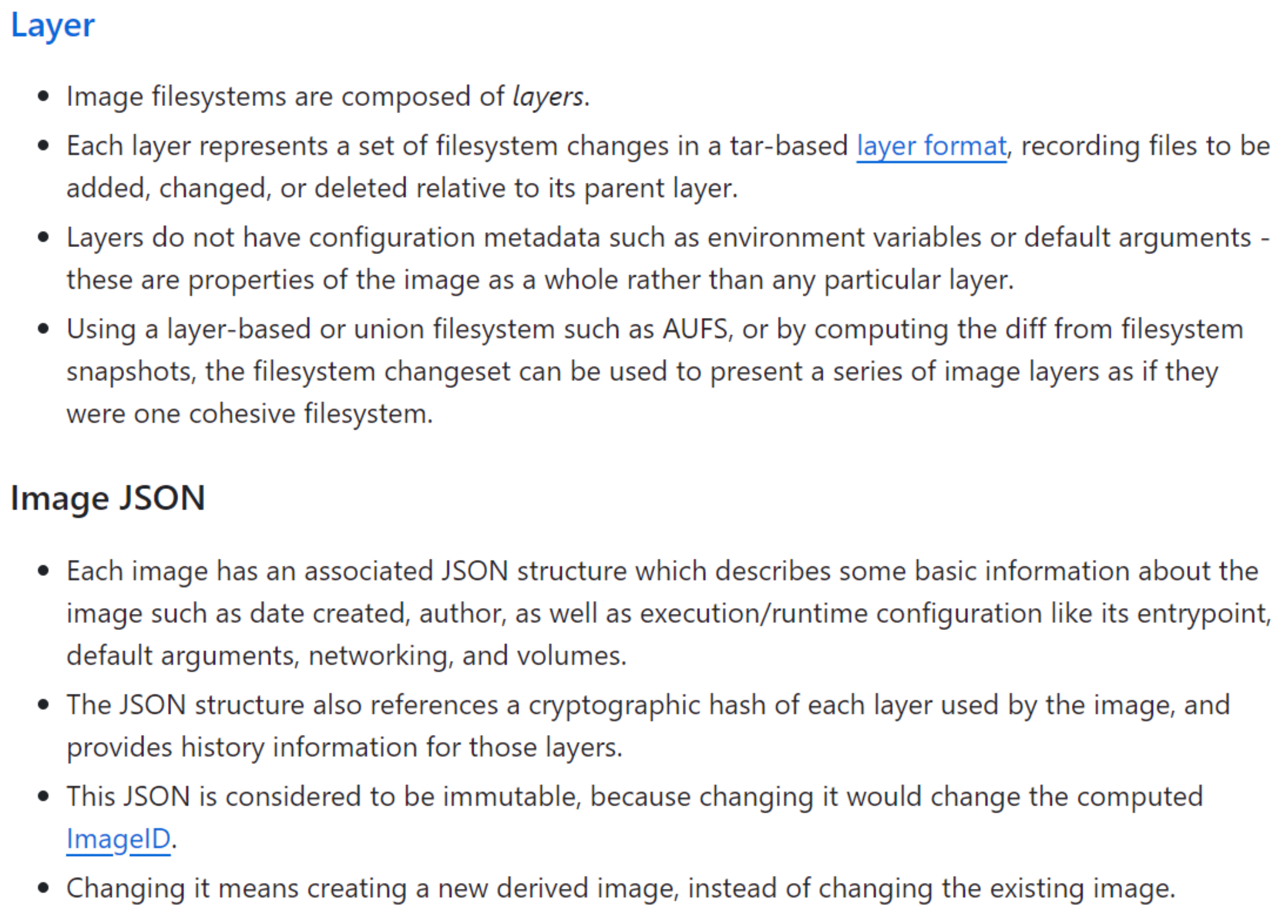
Claim 1	Accused Instrumentalities
	 A container runtime, also known as container engine, is a software component that can run containers on a host operating system. Container runtimes are responsible for loading container images from a repository, monitoring local system resources, isolating system resources for use of a container, and managing container lifecycle. They come in two forms: - High-level container runtimes (such as *containerd* and *CRI-O*) provide functions that run on top of low-level runtime. - Low-level runtimes are responsible for creating and running containers. The primary job of the low-level container runtimes is to provide container lifecycle management. These runtimes implement the Runtime Specification provided by the OCI(Open Container Initiative), a Linux Foundation project started by Docker, which aims to provide open standards for Linux containers. The default reference implementation for low level runtimes specified by OCI is *runc*. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/key-considerations.html

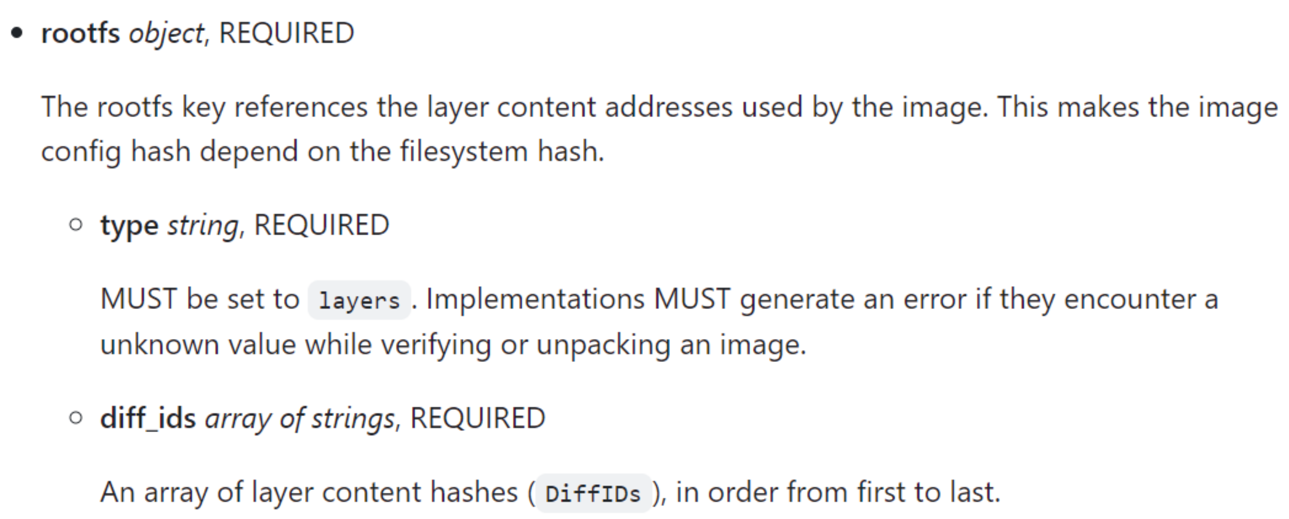
Claim 1	Accused Instrumentalities
	 Open Container Initiative Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

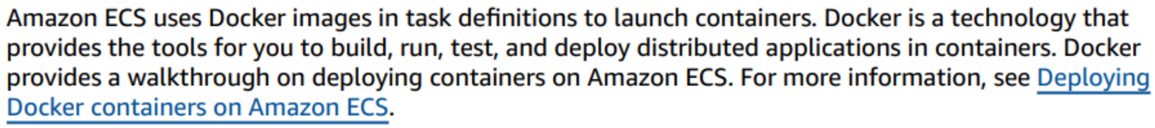
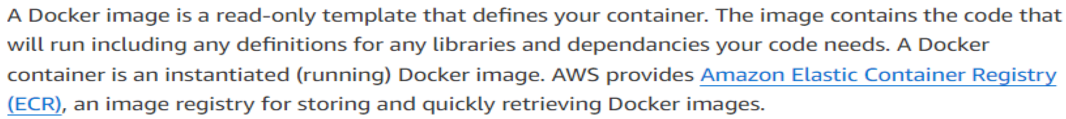
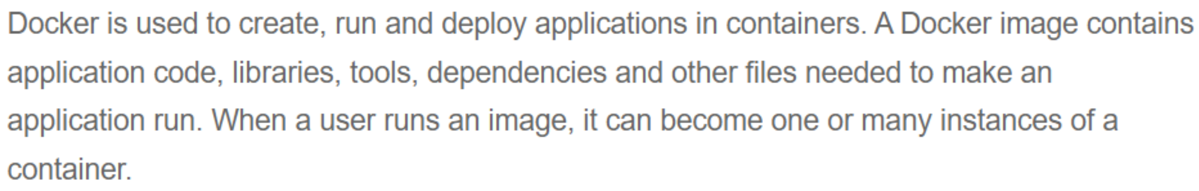
Claim 1	Accused Instrumentalities

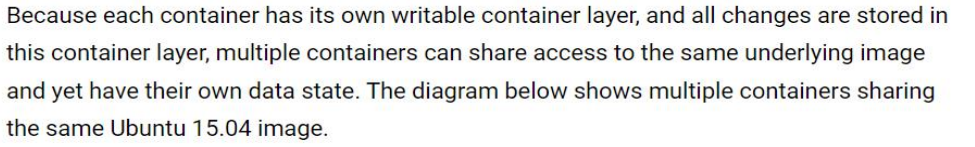
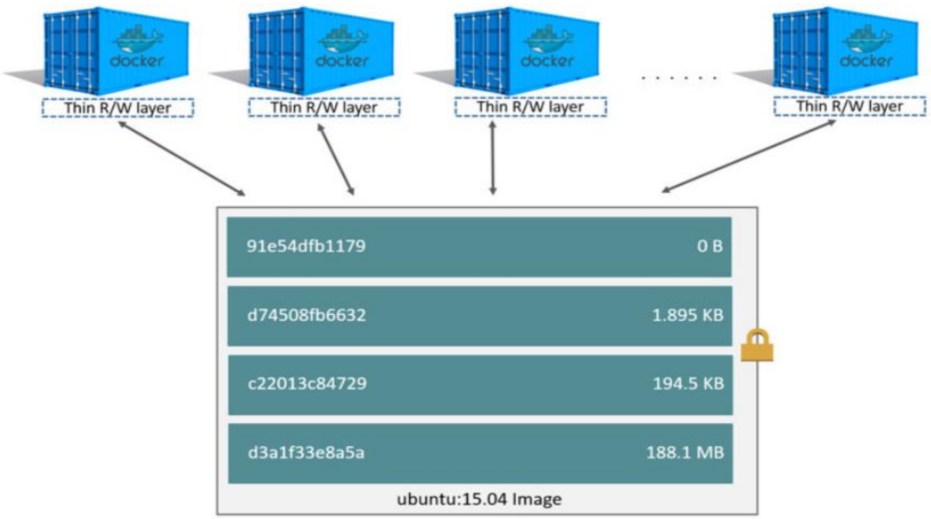
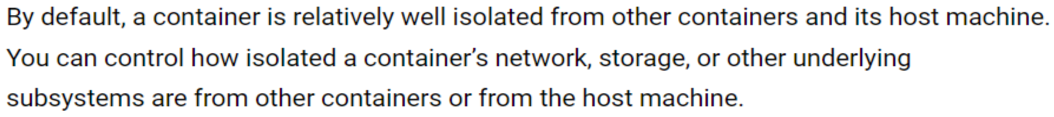
	
	## Overview
	At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.
	`public class HelloWorld {`
	`  public static void main(String[] args) {`
	`    System.out.println("Hello, World");`
	`  }`
	`}`
	 /bin/java
	/opt/app.jar
	/lib/libc
	+
	{
	"manifests": {
	"platform": {
	"os": "linux",
	...
	}
	+
	{
	...
	"config": {
	"Cmd": [
	"java", "-jar",
	"app.jar"
	],
	...
	}
	layer
	image index
	config
	https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

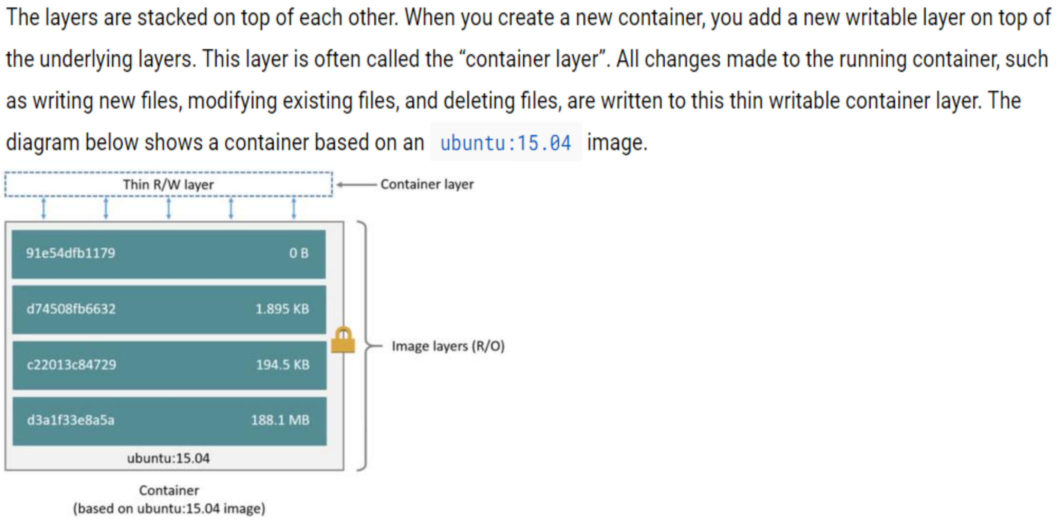
Claim 1	Accused Instrumentalities
	 **OCI Image Configuration** An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

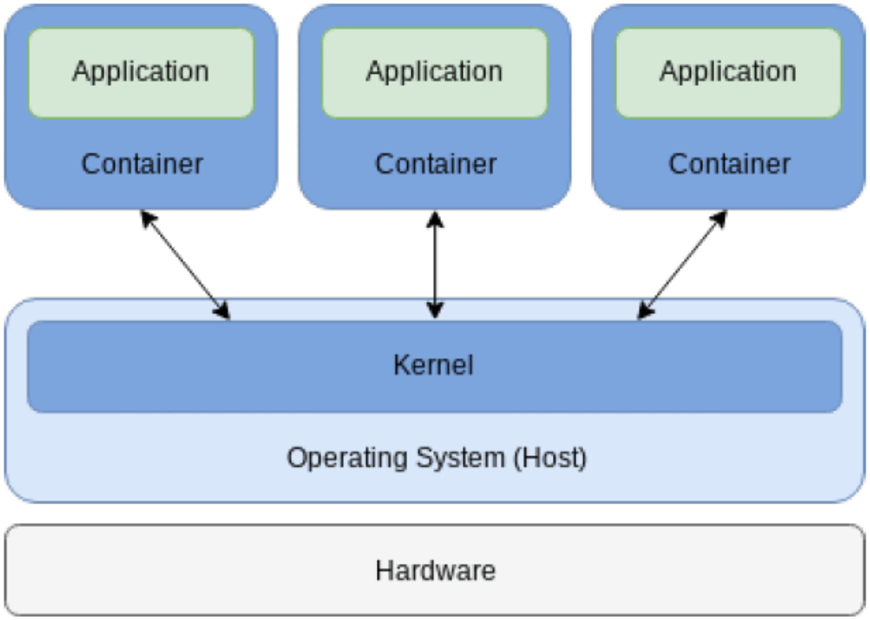
Claim 1	Accused Instrumentalities
	 **Layer** • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

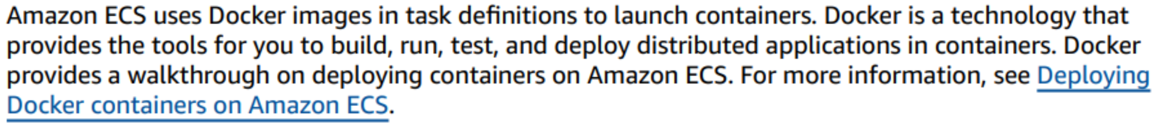
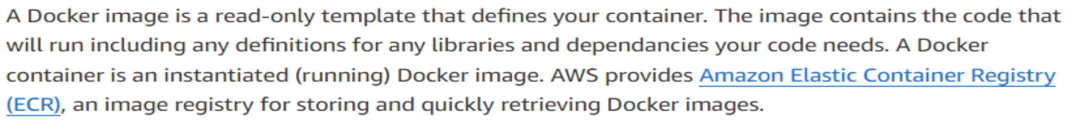
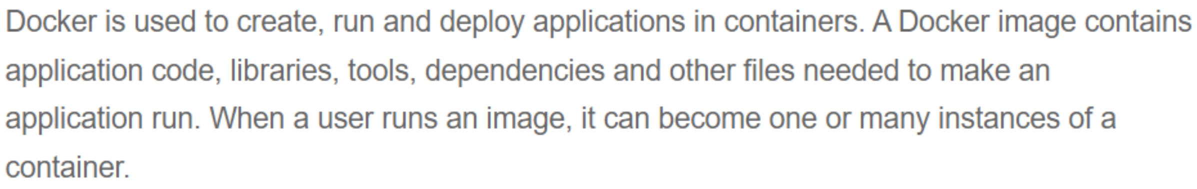
Claim 1	Accused Instrumentalities
	 - **rootfs** *object*, REQUIRED The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash. ○ **type** *string*, REQUIRED MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image. ○ **diff_ids** *array of strings*, REQUIRED An array of layer content hashes (`DiffIDs`), in order from first to last. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md
[1e] ii) wherein an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system	In each Accused Instrumentality, an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system have use of a unique instance of a corresponding critical system element for performing same function. When a Docker or Kubernetes image is used to create a container, it creates a separate and isolated instance of a runtime environment which is independent of other containers running on the same host. Each container has its own instance of base images and its own data. The containers run in isolation, ensuring that the SLCSEs stored in the shared library are accessible to the software applications running in their respective containers. The image includes essential system files, libraries, and dependencies required to run the software application within the container. The containers can share common dependencies and components using layered images. This means that

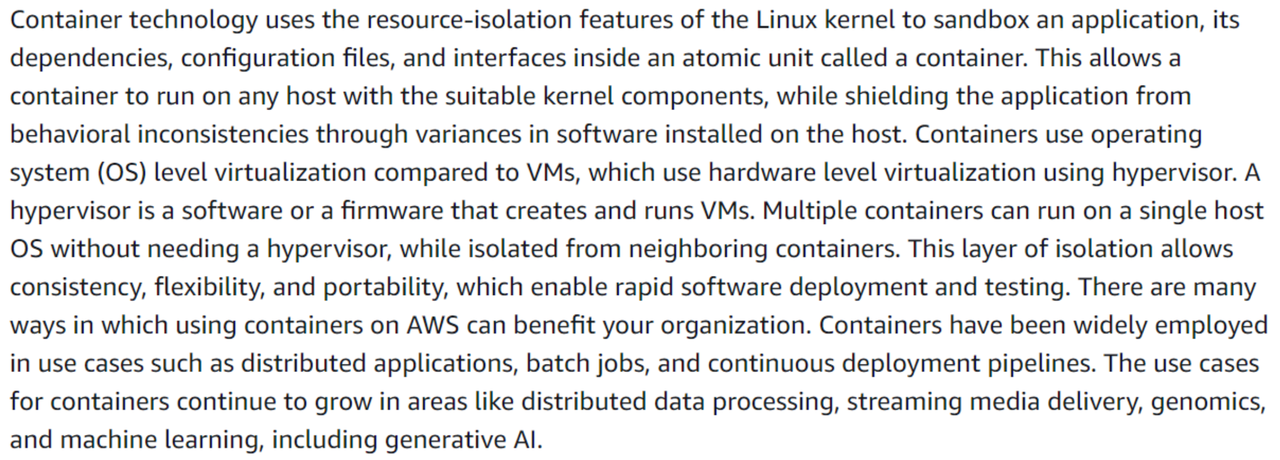
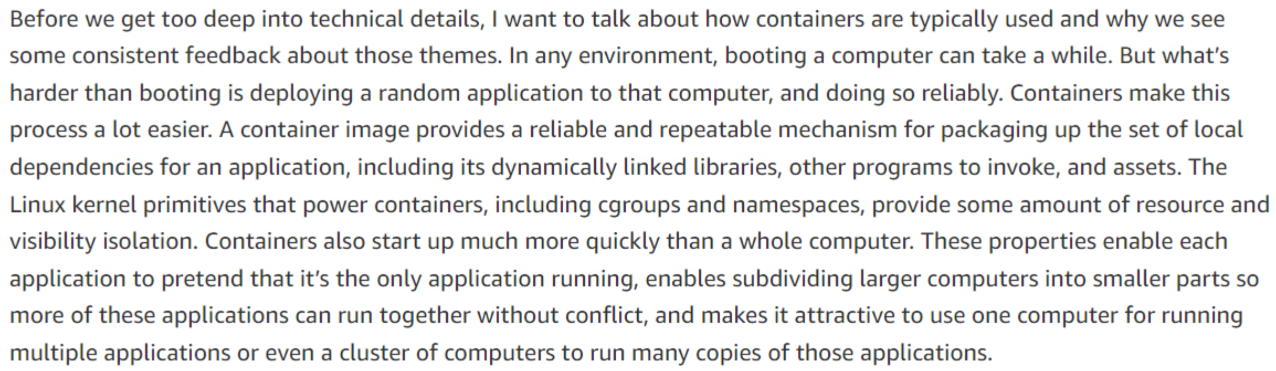
Claim 1	Accused Instrumentalities
have use of a unique instance of a corresponding critical system element for performing same function, and	multiple containers utilize the same base image to create an instance. When an instance of SLCSE is provided from the base image (i.e., from the shared library) to an individual container including application software, it operates in isolation and runs its own instance of the software application without sharing resources or critical system elements with other containers. This ensures that each container has its own isolated context. Docker or Kubernetes containers can share common dependencies and components using layered images. This means that multiple containers can utilize the same base image. Therefore, each container, containing the application software running under the operating system, utilizes a unique instance of the corresponding critical system element to execute the respective application software for performing a same or a different function. *See, e.g.*:  Amazon ECS uses Docker images in task definitions to launch containers. Docker is a technology that provides the tools for you to build, run, test, and deploy distributed applications in containers. Docker provides a walkthrough on deploying containers on Amazon ECS. For more information, see Deploying Docker containers on Amazon ECS. https://docs.aws.amazon.com/pdfs/AmazonECS/latest/developerguide/ecs-dg.pdf  A Docker image is a read-only template that defines your container. The image contains the code that will run including any definitions for any libraries and dependancies your code needs. A Docker container is an instantiated (running) Docker image. AWS provides Amazon Elastic Container Registry (ECR), an image registry for storing and quickly retrieving Docker images. https://aws.amazon.com/docker/#  Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. https://www.techtarget.com/searchitoperations/definition/Docker-image

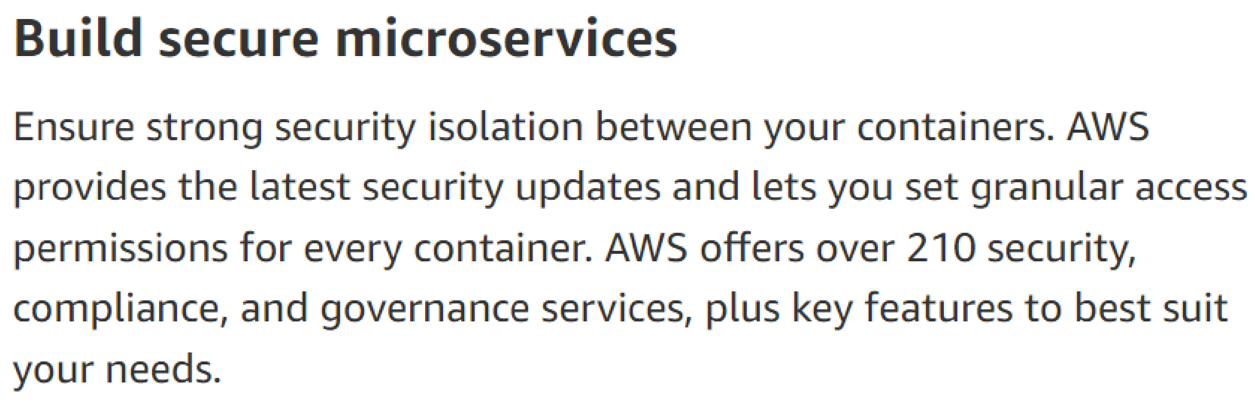
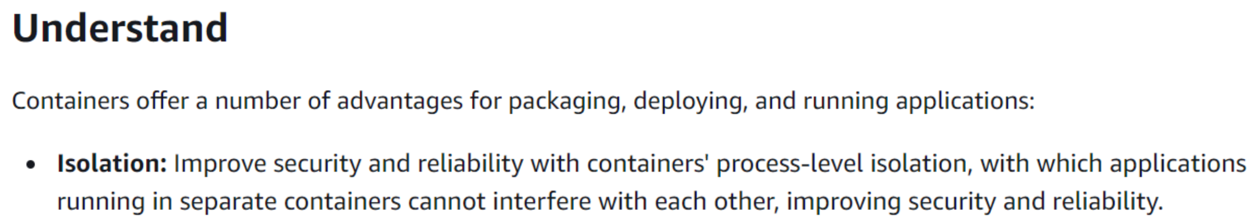
Claim 1	Accused Instrumentalities
	 Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  Thin R/W layer Thin R/W layer Thin R/W layer Thin R/W layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 Image https://docs.docker.com/storage/storagedriver/  By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine. https://docs.docker.com/get-started/overview/

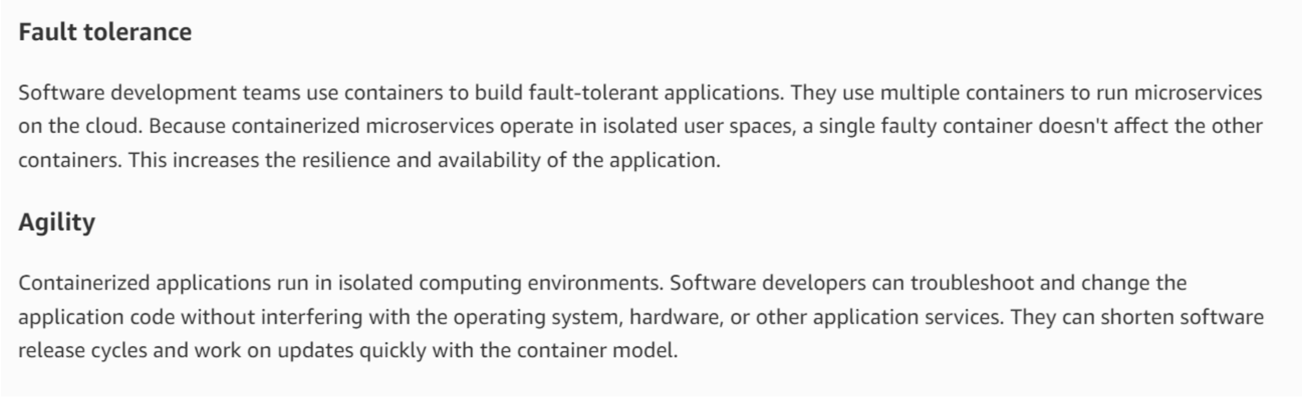
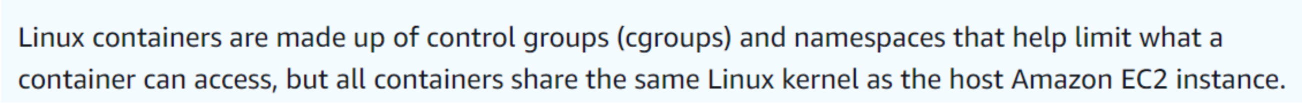
Claim 1	Accused Instrumentalities
	 The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the “container layer”. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image. Thin R/W layer Container layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 Image layers (R/O) Container (based on ubuntu:15.04 image) https://docs.docker.com/storage/storagedriver/

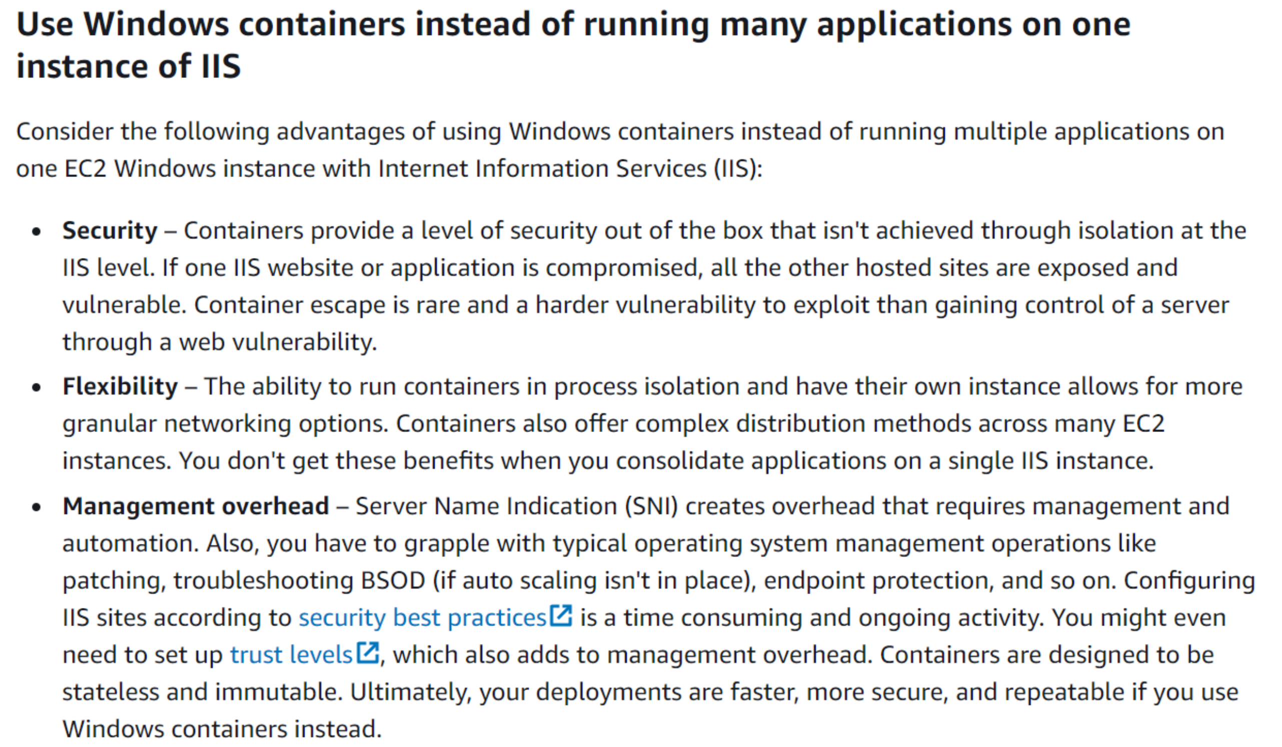
Claim 1	Accused Instrumentalities
	 Application Container Application Container Application Container Kernel Operating System (Host) Hardware https://www.researchgate.net/figure/Docker-container-architecture_fig1_333235708
[1f] iii) wherein a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a	In each Accused Instrumentality, a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously. For example, in Docker or Kubernetes containers, each container operates independently, and a base image includes essential system files, libraries, and dependencies (i.e., SLCSEs) required to run the software application within the container. Based on information and belief, each element, such as system files, libraries, and dependencies (i.e., SLCSE) is associated with an execution of a predetermined function related to the application. When an image is used to create a container in the Accused Instrumentality, an instance of the SLCSE is provided to a software application. Therefore, different instances of the SLCSE are provided to different applications for performing either a same or a different function, simultaneously.

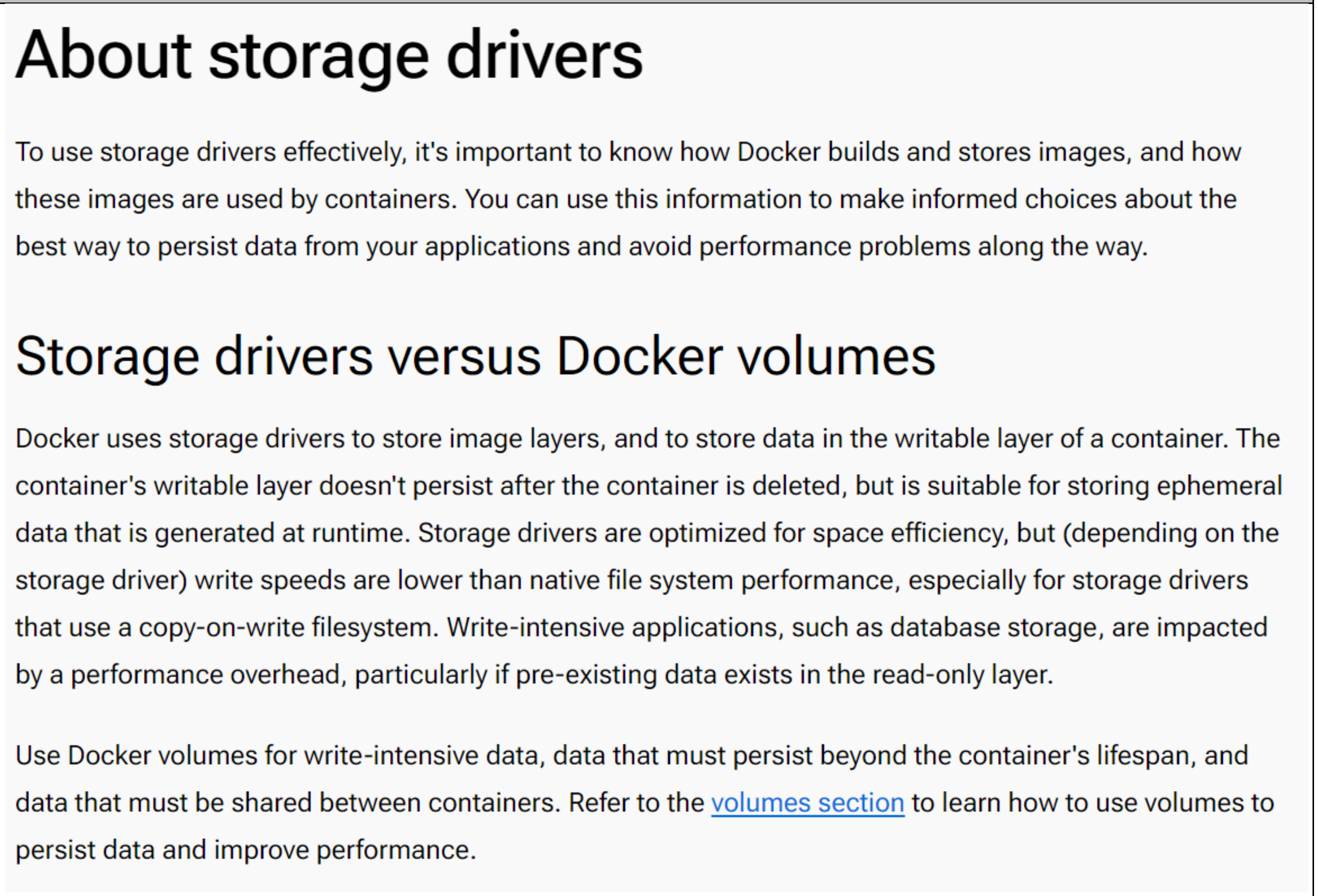
Claim 1	Accused Instrumentalities
second instance of the SLCSE simultaneously.	*See, e.g.*:  Amazon ECS uses Docker images in task definitions to launch containers. Docker is a technology that provides the tools for you to build, run, test, and deploy distributed applications in containers. Docker provides a walkthrough on deploying containers on Amazon ECS. For more information, see Deploying Docker containers on Amazon ECS. https://docs.aws.amazon.com/pdfs/AmazonECS/latest/developerguide/ecs-dg.pdf  A Docker image is a read-only template that defines your container. The image contains the code that will run including any definitions for any libraries and dependancies your code needs. A Docker container is an instantiated (running) Docker image. AWS provides Amazon Elastic Container Registry (ECR), an image registry for storing and quickly retrieving Docker images. https://aws.amazon.com/docker/#  Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. https://www.techtarget.com/searchitoperations/definition/Docker-image

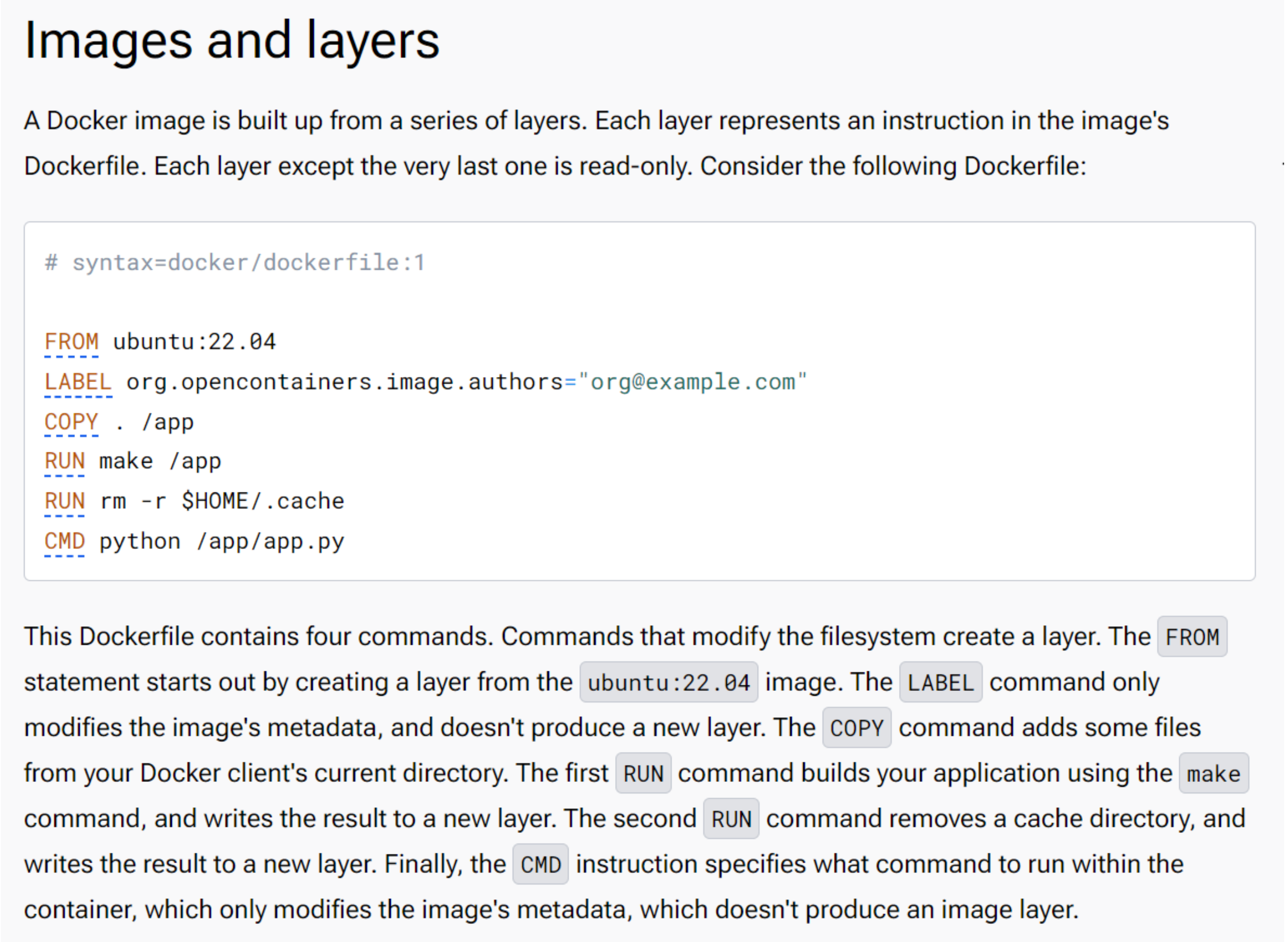
Claim 1	Accused Instrumentalities
	 Container technology uses the resource-isolation features of the Linux kernel to sandbox an application, its dependencies, configuration files, and interfaces inside an atomic unit called a container. This allows a container to run on any host with the suitable kernel components, while shielding the application from behavioral inconsistencies through variances in software installed on the host. Containers use operating system (OS) level virtualization compared to VMs, which use hardware level virtualization using hypervisor. A hypervisor is a software or a firmware that creates and runs VMs. Multiple containers can run on a single host OS without needing a hypervisor, while isolated from neighboring containers. This layer of isolation allows consistency, flexibility, and portability, which enable rapid software deployment and testing. There are many ways in which using containers on AWS can benefit your organization. Containers have been widely employed in use cases such as distributed applications, batch jobs, and continuous deployment pipelines. The use cases for containers continue to grow in areas like distributed data processing, streaming media delivery, genomics, and machine learning, including generative AI. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/containers-on-aws.html  Before we get too deep into technical details, I want to talk about how containers are typically used and why we see some consistent feedback about those themes. In any environment, booting a computer can take a while. But what's harder than booting is deploying a random application to that computer, and doing so reliably. Containers make this process a lot easier. A container image provides a reliable and repeatable mechanism for packaging up the set of local dependencies for an application, including its dynamically linked libraries, other programs to invoke, and assets. The Linux kernel primitives that power containers, including cgroups and namespaces, provide some amount of resource and visibility isolation. Containers also start up much more quickly than a whole computer. These properties enable each application to pretend that it's the only application running, enables subdividing larger computers into smaller parts so more of these applications can run together without conflict, and makes it attractive to use one computer for running multiple applications or even a cluster of computers to run many copies of those applications. https://aws.amazon.com/blogs/containers/bottlerocket-a-special-purpose-container-operating-system/

Claim 1	Accused Instrumentalities
	 **Build secure microservices** Ensure strong security isolation between your containers. AWS provides the latest security updates and lets you set granular access permissions for every container. AWS offers over 210 security, compliance, and governance services, plus key features to best suit your needs. https://aws.amazon.com/containers/  **Understand** Containers offer a number of advantages for packaging, deploying, and running applications: - **Isolation:** Improve security and reliability with containers' process-level isolation, with which applications running in separate containers cannot interfere with each other, improving security and reliability. https://docs.aws.amazon.com/decision-guides/latest/containers-on-aws-how-to-choose/choosing-aws-container-service.html

Claim 1	Accused Instrumentalities
	 **Fault tolerance** Software development teams use containers to build fault-tolerant applications. They use multiple containers to run microservices on the cloud. Because containerized microservices operate in isolated user spaces, a single faulty container doesn't affect the other containers. This increases the resilience and availability of the application. **Agility** Containerized applications run in isolated computing environments. Software developers can troubleshoot and change the application code without interfering with the operating system, hardware, or other application services. They can shorten software release cycles and work on updates quickly with the container model. https://aws.amazon.com/what-is/containerization/  Linux containers are made up of control groups (cgroups) and namespaces that help limit what a container can access, but all containers share the same Linux kernel as the host Amazon EC2 instance. https://docs.aws.amazon.com/eks/latest/userguide/security.html

Claim 1	Accused Instrumentalities
	 **Use Windows containers instead of running many applications on one instance of IIS** Consider the following advantages of using Windows containers instead of running multiple applications on one EC2 Windows instance with Internet Information Services (IIS): • **Security** – Containers provide a level of security out of the box that isn't achieved through isolation at the IIS level. If one IIS website or application is compromised, all the other hosted sites are exposed and vulnerable. Container escape is rare and a harder vulnerability to exploit than gaining control of a server through a web vulnerability. • **Flexibility** – The ability to run containers in process isolation and have their own instance allows for more granular networking options. Containers also offer complex distribution methods across many EC2 instances. You don't get these benefits when you consolidate applications on a single IIS instance. • **Management overhead** – Server Name Indication (SNI) creates overhead that requires management and automation. Also, you have to grapple with typical operating system management operations like patching, troubleshooting BSOD (if auto scaling isn't in place), endpoint protection, and so on. Configuring IIS sites according to security best practices is a time consuming and ongoing activity. You might even need to set up trust levels, which also adds to management overhead. Containers are designed to be stateless and immutable. Ultimately, your deployments are faster, more secure, and repeatable if you use Windows containers instead. https://docs.aws.amazon.com/prescriptive-guidance/latest/optimize-costs-microsoft-workloads/windows-containers-main.html

Claim 1	Accused Instrumentalities
	 **About storage drivers** To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. **Storage drivers versus Docker volumes** Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

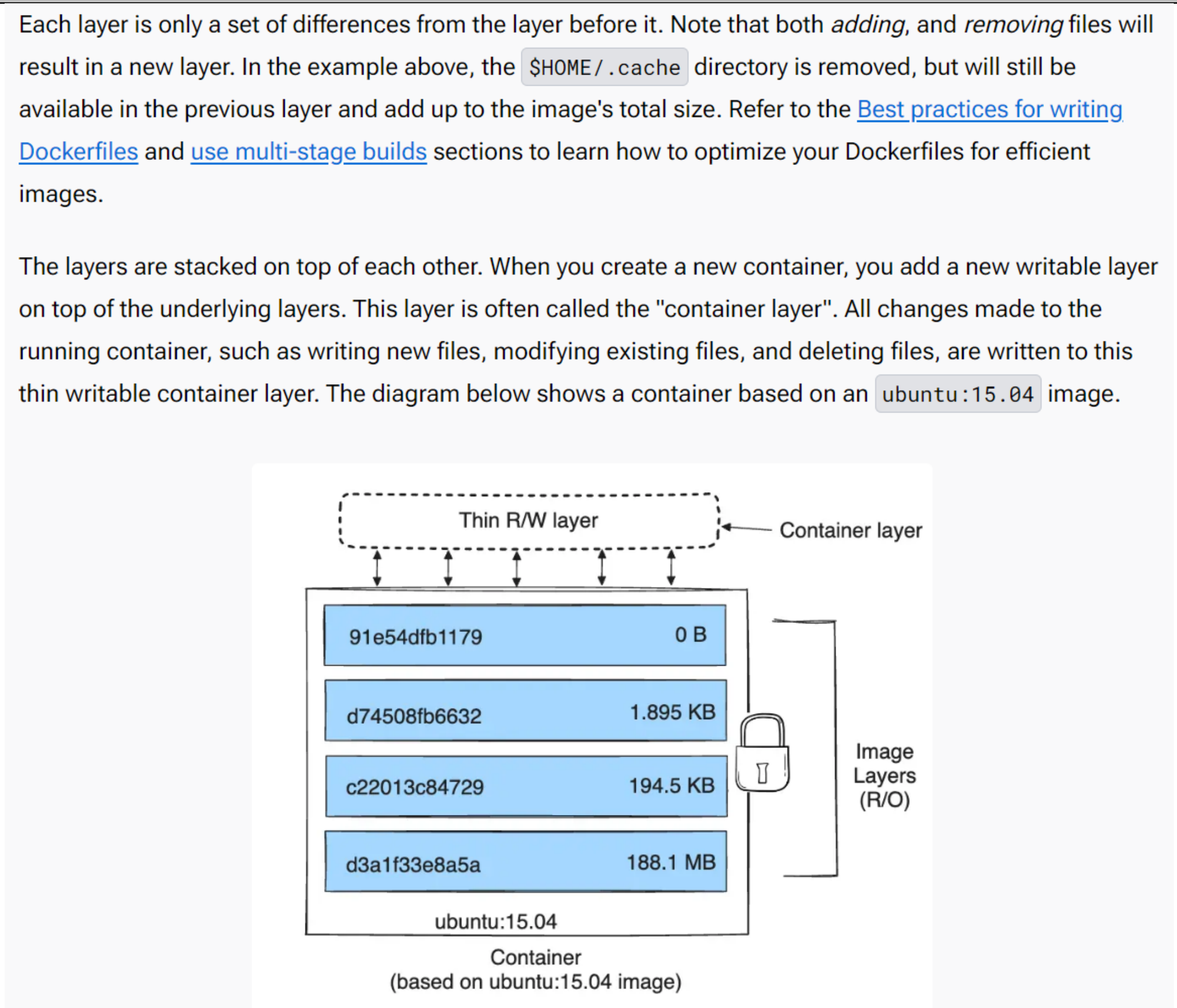
Claim 1	Accused Instrumentalities
	 https://docs.docker.com/storage/storagedriver/

Images and layers

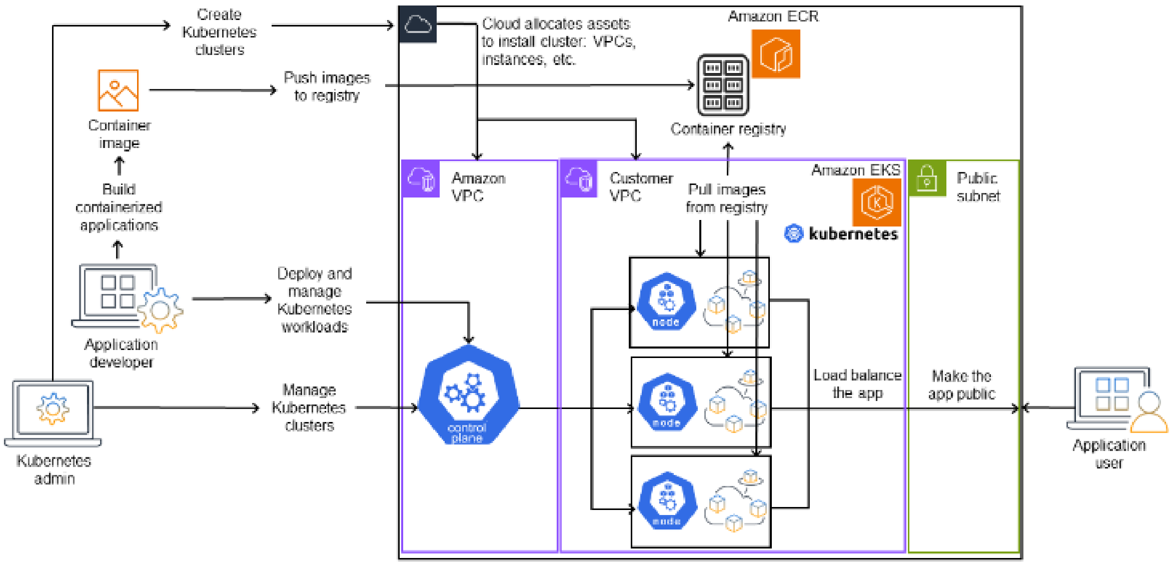
A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

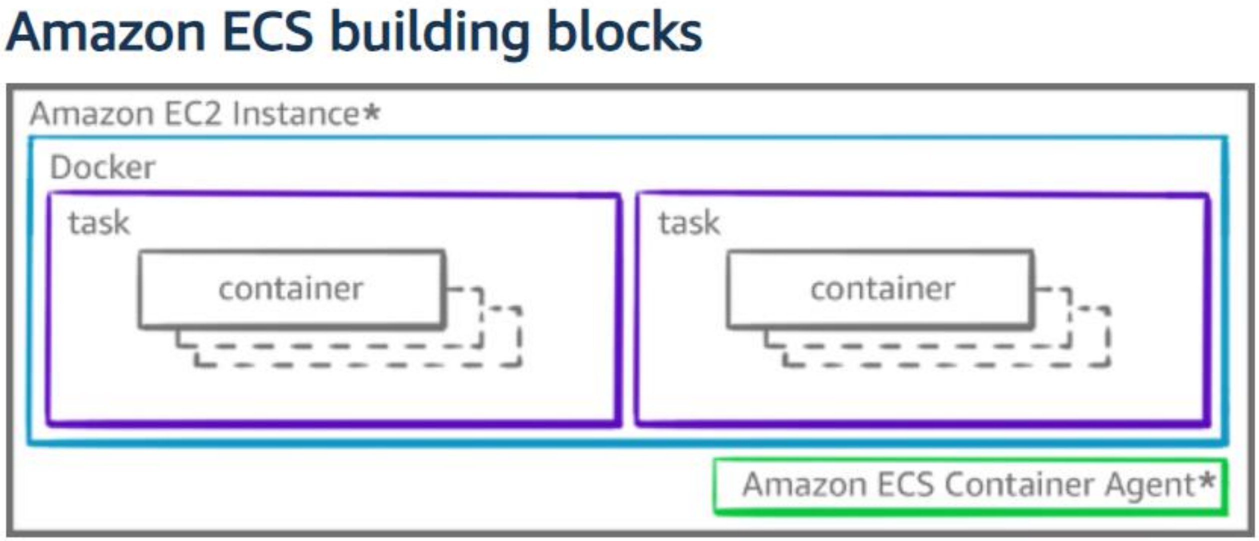
```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

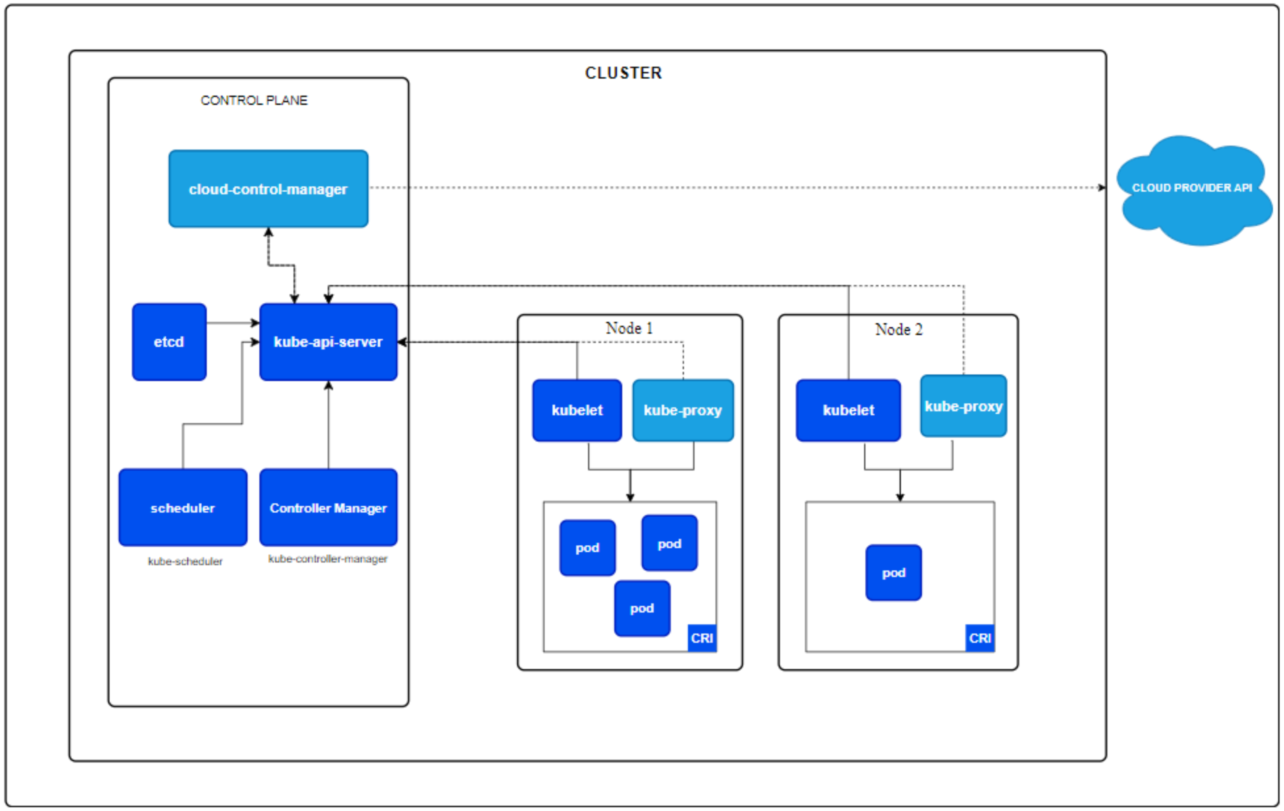
This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

Claim 1	Accused Instrumentalities
	 Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image. Thin R/W layer Container layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 Image Layers (R/O) Container (based on ubuntu:15.04 image) https://docs.docker.com/storage/storagedriver/

Claim 2

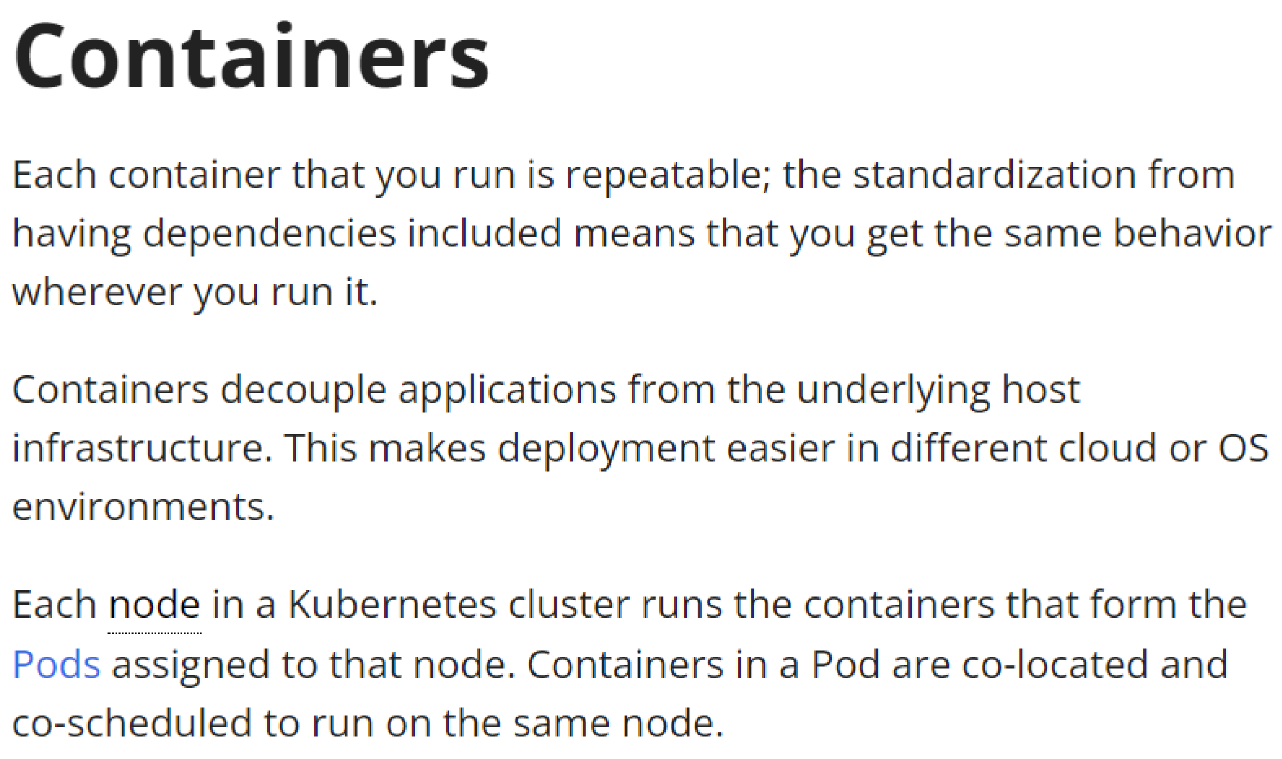
Claim 2	Accused Instrumentalities
2. A computing system as defined in claim 1, wherein in operation, multiple instances of an SLCSE stored in the shared library run simultaneously within the operating system.	Each Accused Instrumentality comprises or constitutes a computing system as defined in claim 1, wherein in operation, multiple instances of an SLCSE stored in the shared library run simultaneously within the operating system. For example, an individual host/node runs multiple containers and/or pods simultaneously, each of which has an instance of an SLCSE. *See, e.g.*: A Kubernetes cluster in action  Create Kubernetes clusters Cloud allocates assets to install cluster: VPCs, instances, etc. Amazon ECR Push images to registry Container image Container registry Build containerized applications Amazon VPC Customer VPC Pull images from registry Amazon EKS kubernetes Public subnet Deploy and manage Kubernetes workloads Application developer node node node control plane Load balance the app Make the app public Manage Kubernetes clusters Kubernetes admin Application user https://docs.aws.amazon.com/eks/latest/userguide/kubernetes-concepts.html

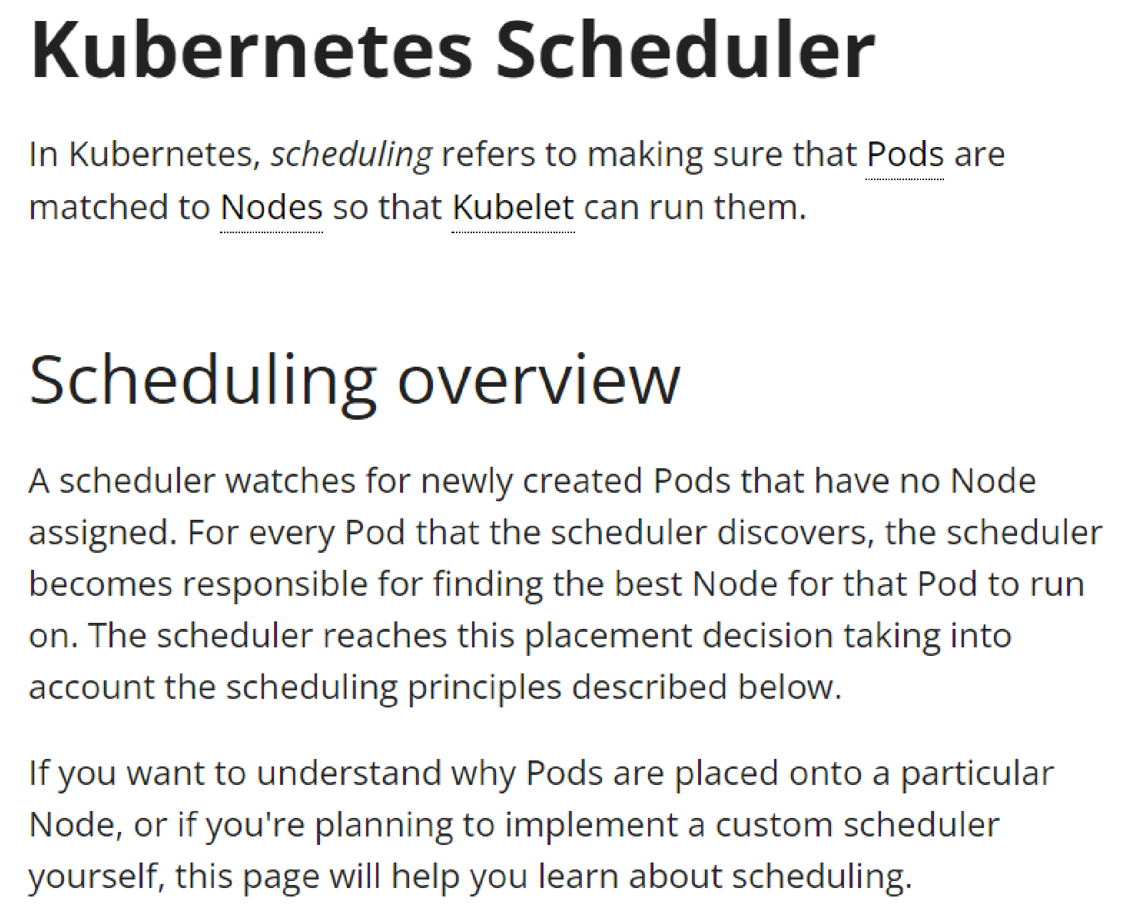
Claim 2	Accused Instrumentalities
	 Amazon ECS building blocks Amazon EC2 Instance* Docker task container task container Amazon ECS Container Agent* https://aws.amazon.com/blogs/compute/building-blocks-of-amazon-ecs/

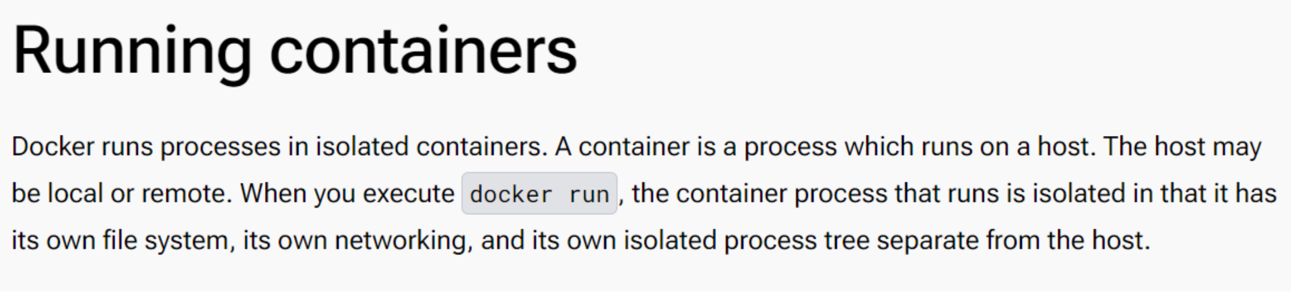
<table>
<tr><th>Claim 2</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>


Kubernetes cluster architecture

https://kubernetes.io/docs/concepts/architecture/
</td></tr>
</table>

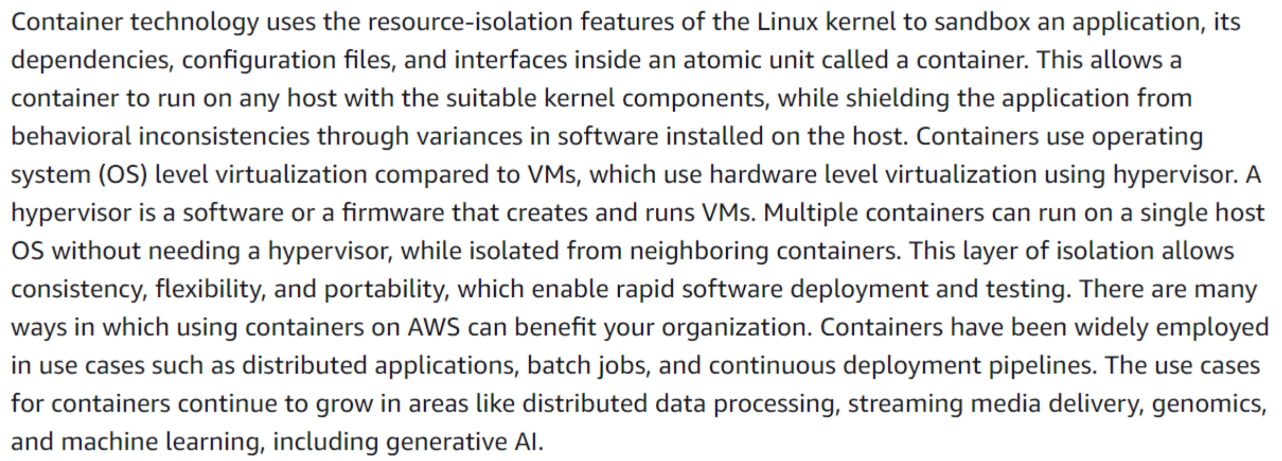
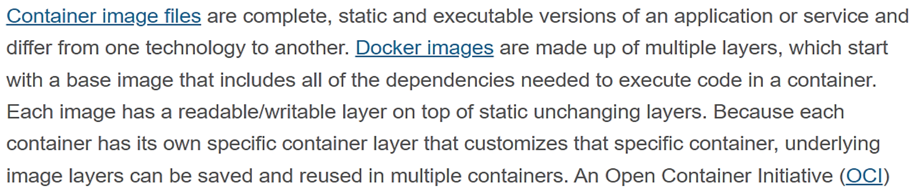
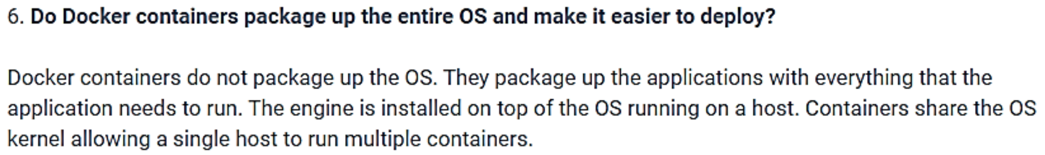
Claim 2	Accused Instrumentalities
	 **Containers** Each container that you run is repeatable; the standardization from having dependencies included means that you get the same behavior wherever you run it. Containers decouple applications from the underlying host infrastructure. This makes deployment easier in different cloud or OS environments. Each node in a Kubernetes cluster runs the containers that form the Pods assigned to that node. Containers in a Pod are co-located and co-scheduled to run on the same node. https://kubernetes.io/docs/concepts/containers/

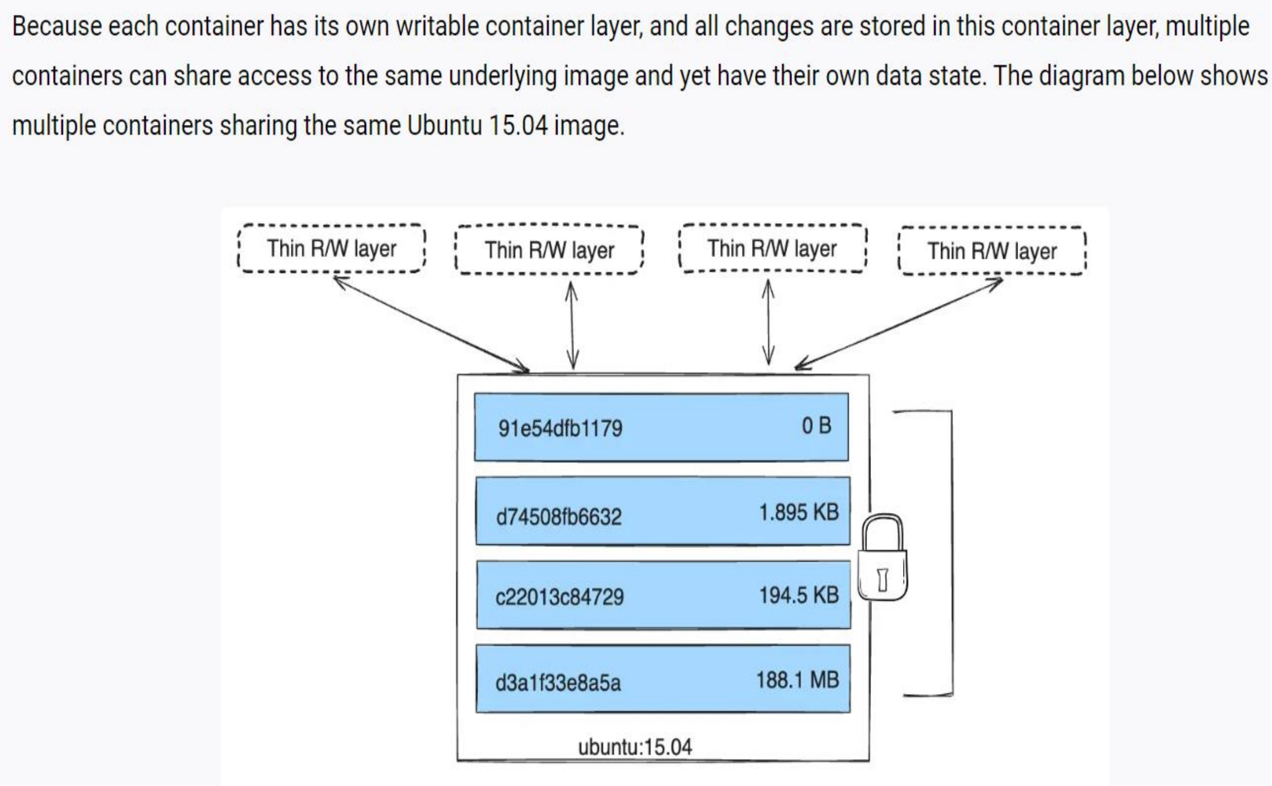
Claim 2	Accused Instrumentalities
	 **Kubernetes Scheduler** In Kubernetes, *scheduling* refers to making sure that Pods are matched to Nodes so that Kubelet can run them. Scheduling overview A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on. The scheduler reaches this placement decision taking into account the scheduling principles described below. If you want to understand why Pods are placed onto a particular Node, or if you're planning to implement a custom scheduler yourself, this page will help you learn about scheduling. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

Claim 2	Accused Instrumentalities
	 Running containers Docker runs processes in isolated containers. A container is a process which runs on a host. The host may be local or remote. When you execute `docker run`, the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host. https://docs.docker.com/engine/reference/run/

Claim 3

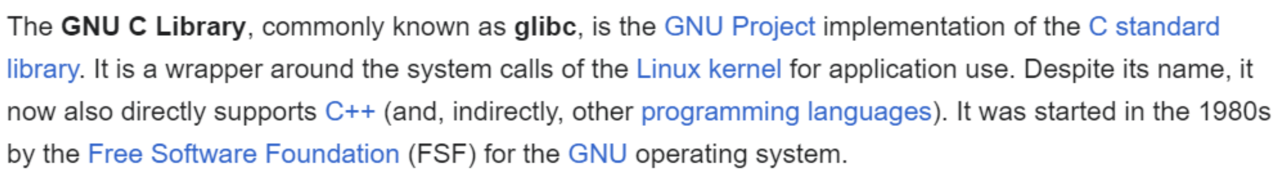
Claim 3	Accused Instrumentalities
3. A computing system according to claim 1 wherein OSCSEs corresponding to and capable of performing the same function as SLCSEs remain in the operating system kernel.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein OSCSEs corresponding to and capable of performing the same function as SLCSEs remain in the operating system kernel. For example, both Docker and Kubernetes systems preserve the host kernel substantially unchanged; therefore the OSCSEs corresponding to the SLCSEs remain in the operating system kernel. *See, e.g.*:

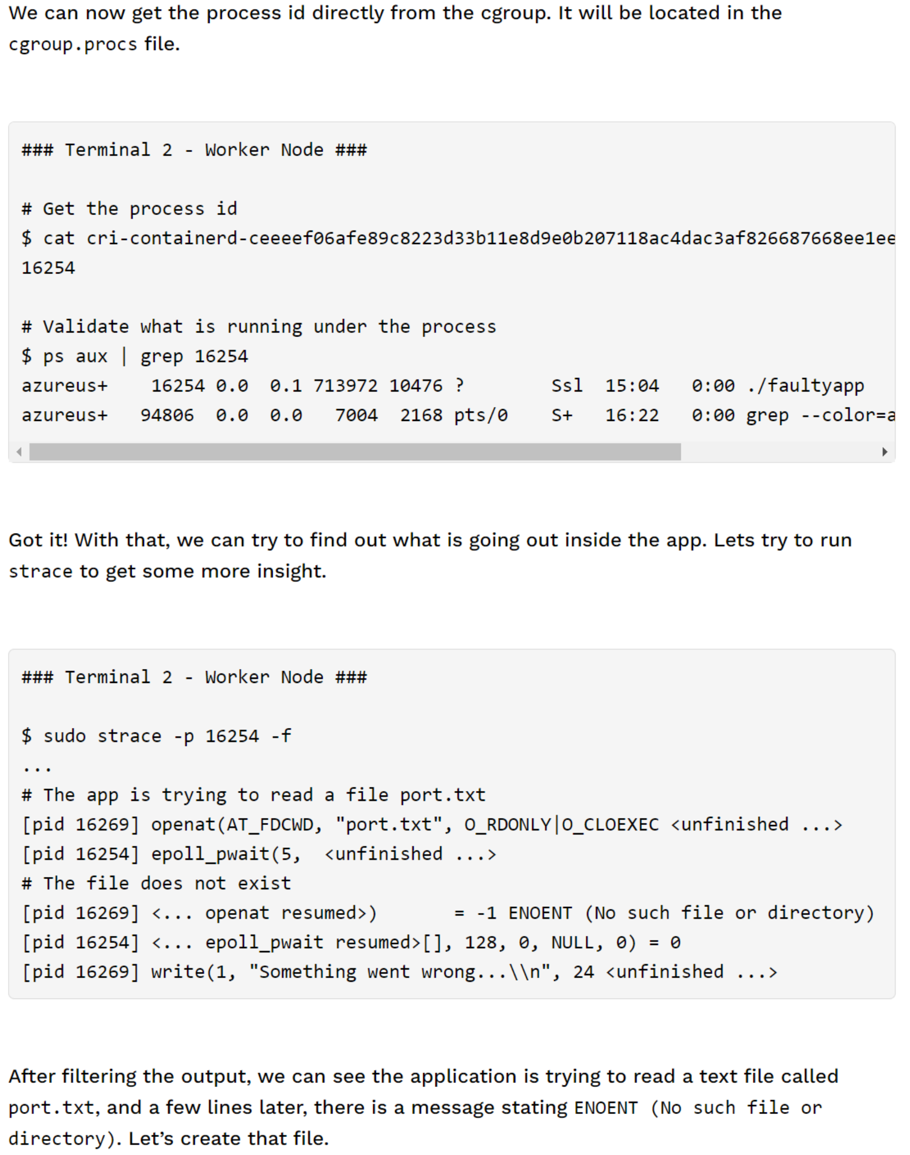
Claim 3	Accused Instrumentalities
	 Container technology uses the resource-isolation features of the Linux kernel to sandbox an application, its dependencies, configuration files, and interfaces inside an atomic unit called a container. This allows a container to run on any host with the suitable kernel components, while shielding the application from behavioral inconsistencies through variances in software installed on the host. Containers use operating system (OS) level virtualization compared to VMs, which use hardware level virtualization using hypervisor. A hypervisor is a software or a firmware that creates and runs VMs. Multiple containers can run on a single host OS without needing a hypervisor, while isolated from neighboring containers. This layer of isolation allows consistency, flexibility, and portability, which enable rapid software deployment and testing. There are many ways in which using containers on AWS can benefit your organization. Containers have been widely employed in use cases such as distributed applications, batch jobs, and continuous deployment pipelines. The use cases for containers continue to grow in areas like distributed data processing, streaming media delivery, genomics, and machine learning, including generative AI. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/containers-on-aws.html  Container image files are complete, static and executable versions of an application or service and differ from one technology to another. Docker images are made up of multiple layers, which start with a base image that includes all of the dependencies needed to execute code in a container. Each image has a readable/writable layer on top of static unchanging layers. Because each container has its own specific container layer that customizes that specific container, underlying image layers can be saved and reused in multiple containers. An Open Container Initiative (OCI) https://www.techtarget.com/searchitoperations/definition/container-containerization-or-container-based-virtualization  6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/

Claim 3	Accused Instrumentalities
	 Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image. Thin R/W layer Thin R/W layer Thin R/W layer Thin R/W layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 https://docs.docker.com/storage/storagedriver/

Claim 4

Claim 4	Accused Instrumentalities
4. A computing system according to claim 1 wherein the one or more SLCSEs provided to one of the plurality of software applications having exclusive use thereof,	Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein the one or more SLCSEs provided to one of the plurality of software applications having exclusive use thereof, use system calls to access services in the operating system kernel. For example, the SLCSEs in a container use system calls to access services in the operating system kernel. For example, the glibc library (or other similar library) in the container uses system calls to

Claim 4	Accused Instrumentalities
use system calls to access services in the operating system kernel.	interface with the host Linux kernel. In general, system calls can be observed using a tool such as strace. *See, e.g.*:  The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc



We can now get the process id directly from the cgroup. It will be located in the `cgroup.procs` file.

```
### Terminal 2 - Worker Node ###

# Get the process id
$ cat cri-containerd-ceeeef06afe89c8223d33b11e8d9e0b207118ac4dac3af826687668ee1ee
16254

# Validate what is running under the process
$ ps aux | grep 16254
azureus+    16254 0.0  0.1 713972 10476 ?        Ssl  15:04   0:00 ./faultyapp
azureus+   94806  0.0  0.0   7004  2168 pts/0    S+   16:22   0:00 grep --color=a
```

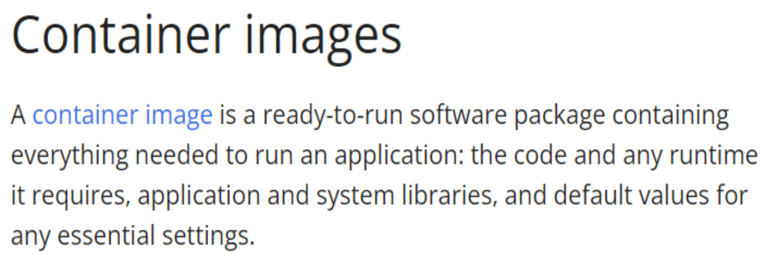
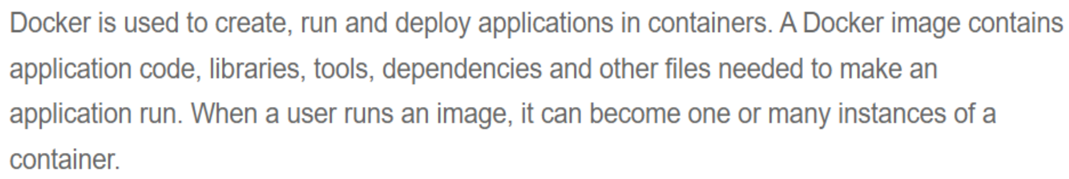
Got it! With that, we can try to find out what is going out inside the app. Lets try to run `strace` to get some more insight.

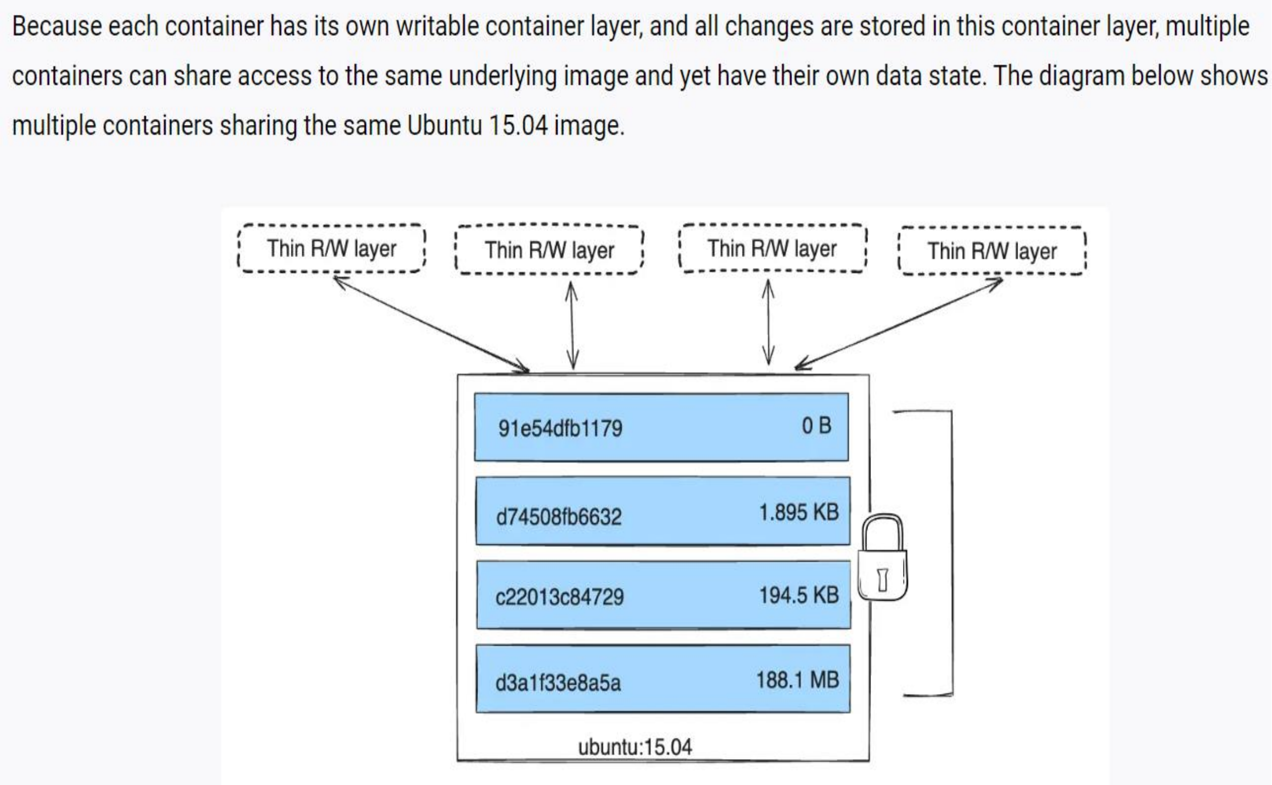
```
### Terminal 2 - Worker Node ###

$ sudo strace -p 16254 -f
...
# The app is trying to read a file port.txt
[pid 16269] openat(AT_FDCWD, "port.txt", O_RDONLY|O_CLOEXEC <unfinished ...>
[pid 16254] epoll_pwait(5,  <unfinished ...>
# The file does not exist
[pid 16269] <... openat resumed>)       = -1 ENOENT (No such file or directory)
[pid 16254] <... epoll_pwait resumed>[], 128, 0, NULL, 0) = 0
[pid 16269] write(1, "Something went wrong...\\n", 24 <unfinished ...>
```

After filtering the output, we can see the application is trying to read a text file called `port.txt`, and a few lines later, there is a message stating `ENOENT (No such file or directory)`. Let's create that file.

https://www.berops.com/blog/a-different-method-to-debug-kubernetes-pods

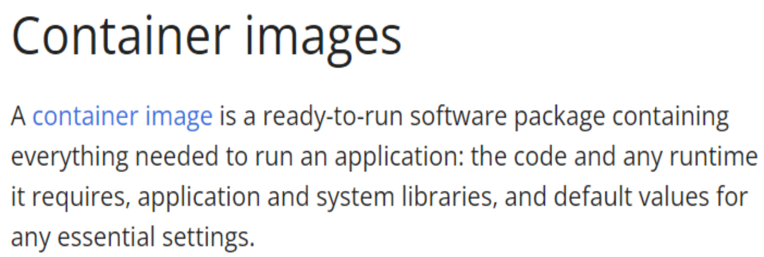
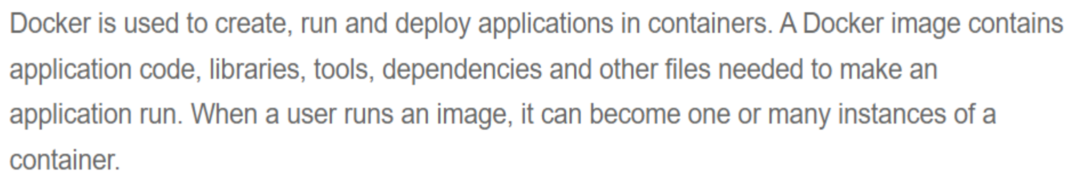
Claim 5

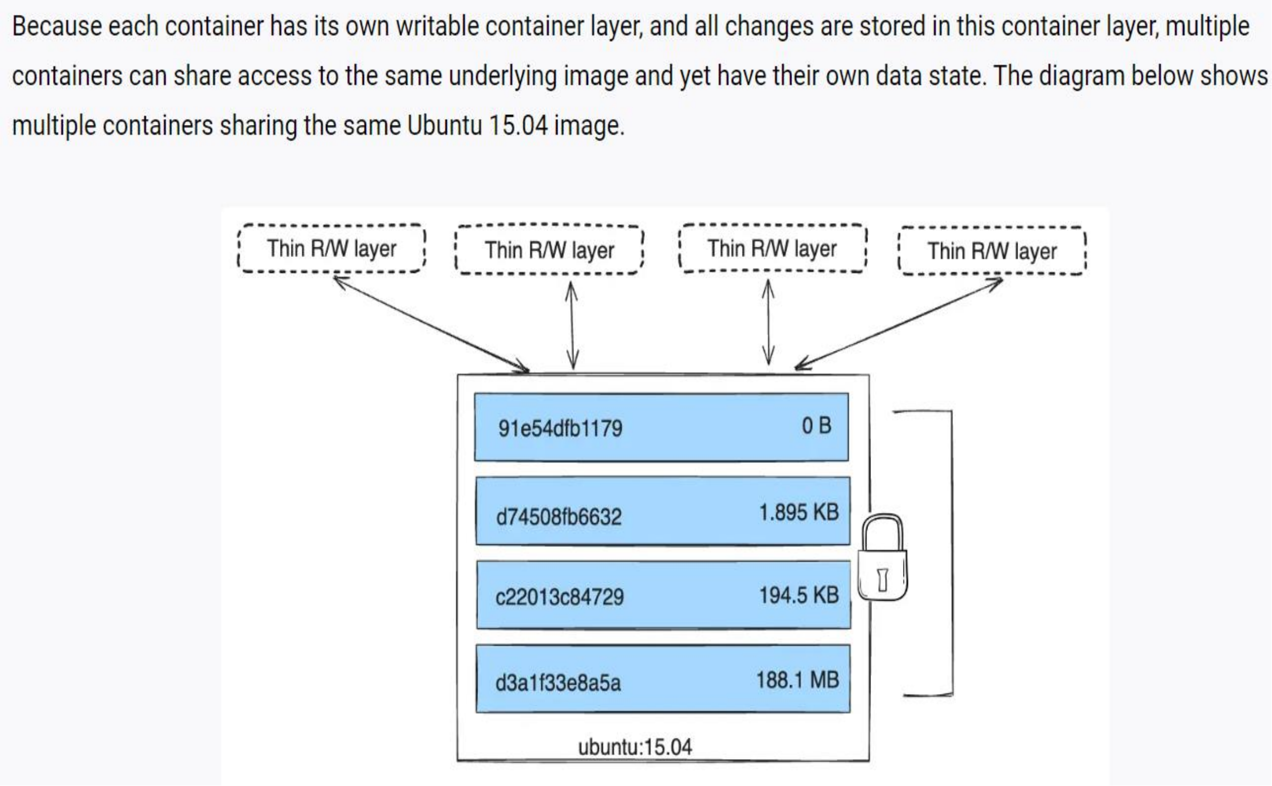
Claim 5	Accused Instrumentalities
5. A computing system according to claim 1 wherein the operating system kernel comprises a kernel module adapted to serve as an interface between an SLCSE in the context of an application program and a device driver.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein the operating system kernel comprises a kernel module adapted to serve as an interface between an SLCSE in the context of an application program and a device driver. For example, the server (node) includes an operating system having a kernel. The kernel comprises a kernel module which enables applications (including their libraries) to have access to system resources such as storage, *i.e.*, acts as an interface between applications/libraries and OS libraries or drivers *See, e.g.*:  Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/  Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. ttps://www.techtarget.com/searchitoperations/definition/Docker-image

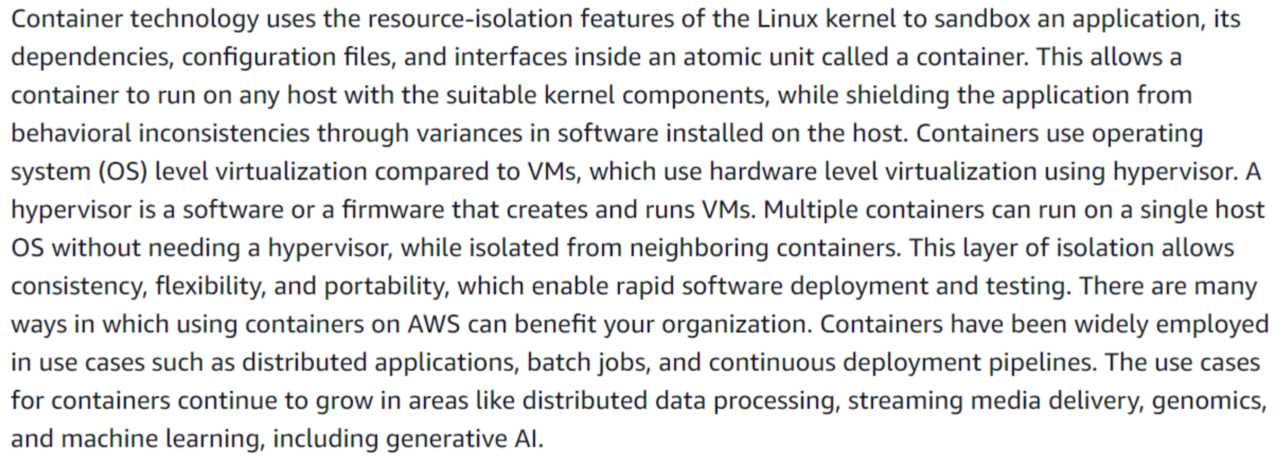
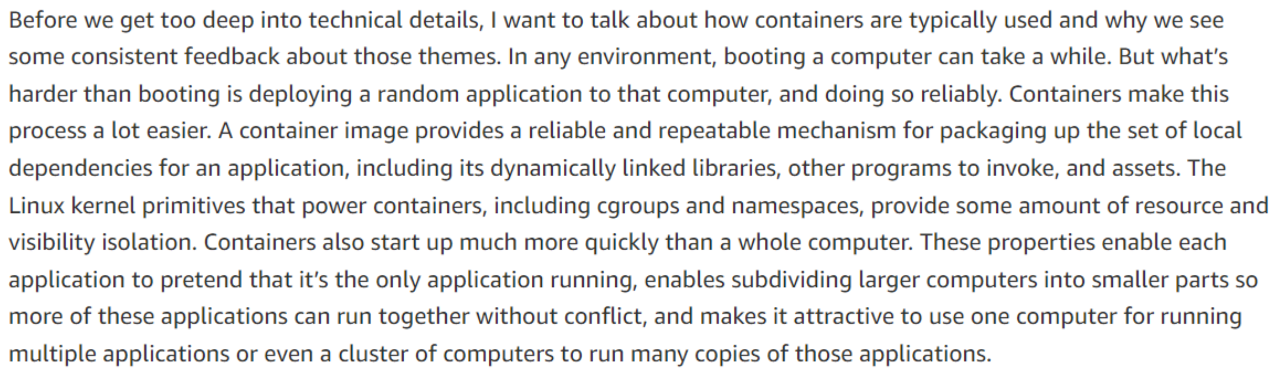
Claim 5	Accused Instrumentalities
	 Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image. Thin R/W layer Thin R/W layer Thin R/W layer Thin R/W layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 https://docs.docker.com/storage/storagedriver/

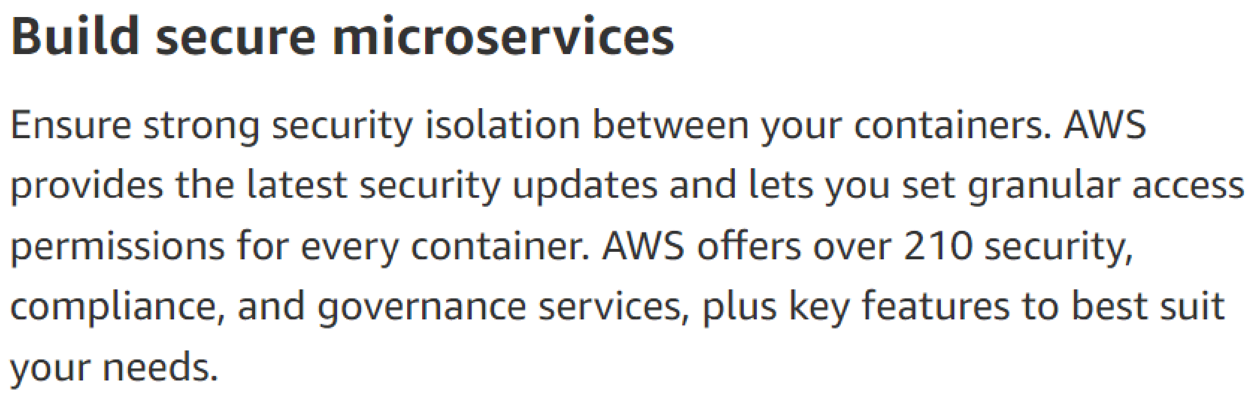
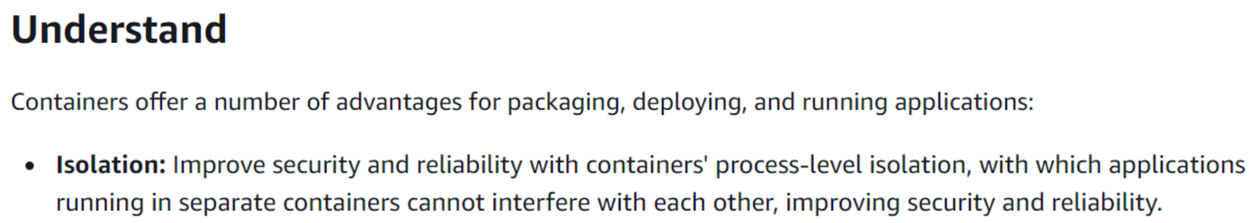
Claim 10

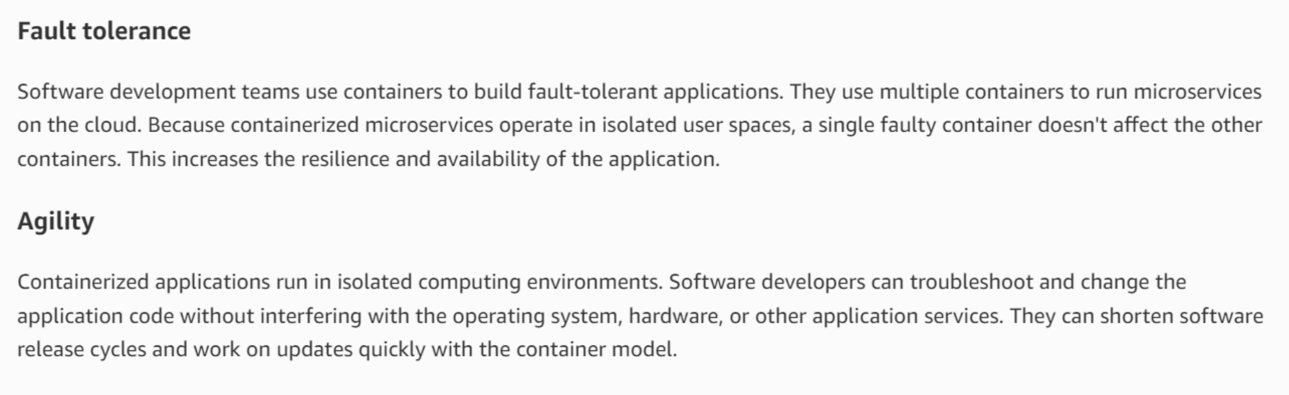
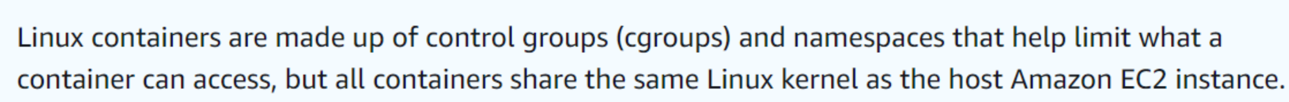
Claim 10	Accused Instrumentalities
10. A computing system according to claim 2 wherein SLCSEs stored in the shared library are linked to particular software applications of the	Each Accused Instrumentality comprises or constitutes a computing system according to claim 2 wherein SLCSEs stored in the shared library are linked to particular software applications of the plurality of software applications as the particular software applications are loaded such that the particular software applications have a link that provides unique access to a unique instance of a CSE.

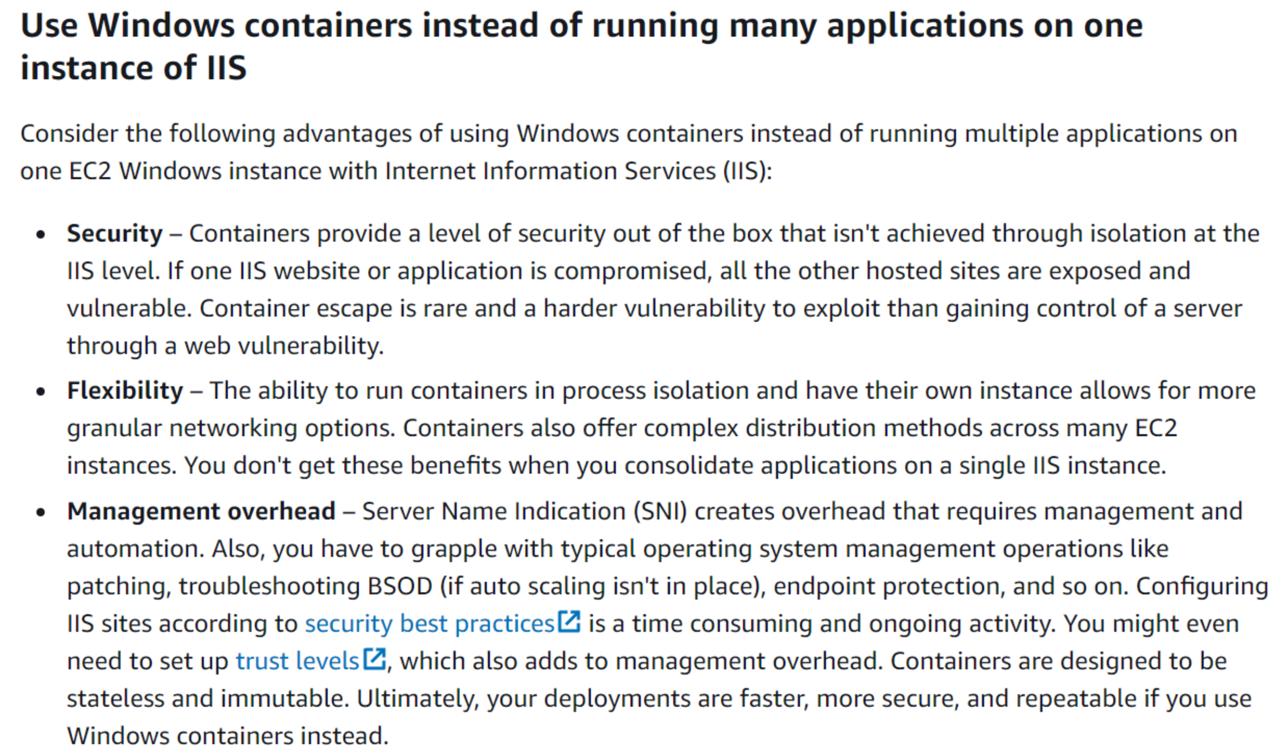
Claim 10	Accused Instrumentalities
plurality of software applications as the particular software applications are loaded such that the particular software applications have a link that provides unique access to a unique instance of a CSE.	For example, the containers can share common dependencies and components using layered images, and multiple containers can use the same base image. Therefore, each container, containing the application software running under the operating system of the server hosting a Kubernetes pod, uses a unique instance of the corresponding critical system element to execute the respective application software and has a link to that unique instance. *See, e.g.*:  Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/  Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. ttps://www.techtarget.com/searchitoperations/definition/Docker-image

Claim 10	Accused Instrumentalities
	 Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image. Thin R/W layer Thin R/W layer Thin R/W layer Thin R/W layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 https://docs.docker.com/storage/storagedriver/

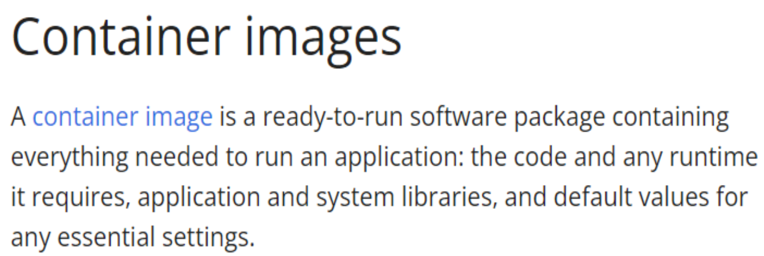
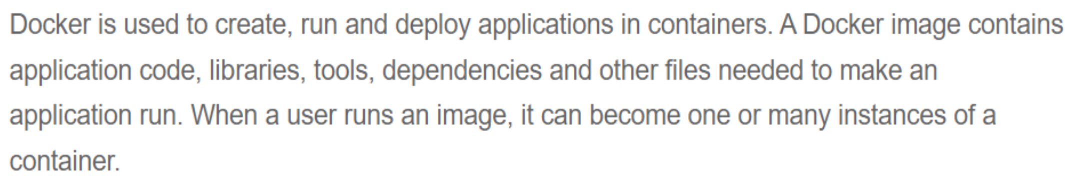
Claim 10	Accused Instrumentalities
	 Container technology uses the resource-isolation features of the Linux kernel to sandbox an application, its dependencies, configuration files, and interfaces inside an atomic unit called a container. This allows a container to run on any host with the suitable kernel components, while shielding the application from behavioral inconsistencies through variances in software installed on the host. Containers use operating system (OS) level virtualization compared to VMs, which use hardware level virtualization using hypervisor. A hypervisor is a software or a firmware that creates and runs VMs. Multiple containers can run on a single host OS without needing a hypervisor, while isolated from neighboring containers. This layer of isolation allows consistency, flexibility, and portability, which enable rapid software deployment and testing. There are many ways in which using containers on AWS can benefit your organization. Containers have been widely employed in use cases such as distributed applications, batch jobs, and continuous deployment pipelines. The use cases for containers continue to grow in areas like distributed data processing, streaming media delivery, genomics, and machine learning, including generative AI. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/containers-on-aws.html  Before we get too deep into technical details, I want to talk about how containers are typically used and why we see some consistent feedback about those themes. In any environment, booting a computer can take a while. But what's harder than booting is deploying a random application to that computer, and doing so reliably. Containers make this process a lot easier. A container image provides a reliable and repeatable mechanism for packaging up the set of local dependencies for an application, including its dynamically linked libraries, other programs to invoke, and assets. The Linux kernel primitives that power containers, including cgroups and namespaces, provide some amount of resource and visibility isolation. Containers also start up much more quickly than a whole computer. These properties enable each application to pretend that it's the only application running, enables subdividing larger computers into smaller parts so more of these applications can run together without conflict, and makes it attractive to use one computer for running multiple applications or even a cluster of computers to run many copies of those applications. https://aws.amazon.com/blogs/containers/bottlerocket-a-special-purpose-container-operating-system/

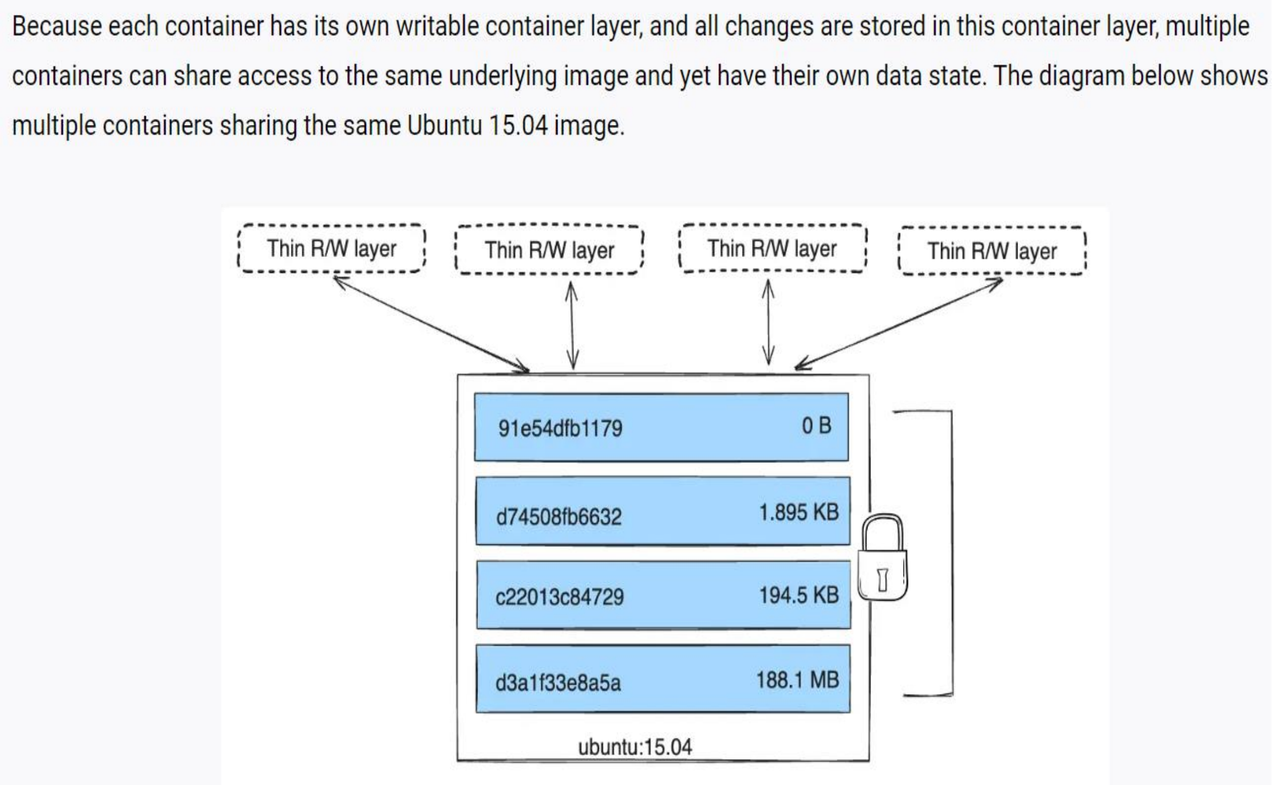
Claim 10	Accused Instrumentalities
	 **Build secure microservices** Ensure strong security isolation between your containers. AWS provides the latest security updates and lets you set granular access permissions for every container. AWS offers over 210 security, compliance, and governance services, plus key features to best suit your needs. https://aws.amazon.com/containers/  **Understand** Containers offer a number of advantages for packaging, deploying, and running applications: • **Isolation:** Improve security and reliability with containers' process-level isolation, with which applications running in separate containers cannot interfere with each other, improving security and reliability. https://docs.aws.amazon.com/decision-guides/latest/containers-on-aws-how-to-choose/choosing-aws-container-service.html

Claim 10	Accused Instrumentalities
	 **Fault tolerance** Software development teams use containers to build fault-tolerant applications. They use multiple containers to run microservices on the cloud. Because containerized microservices operate in isolated user spaces, a single faulty container doesn't affect the other containers. This increases the resilience and availability of the application. **Agility** Containerized applications run in isolated computing environments. Software developers can troubleshoot and change the application code without interfering with the operating system, hardware, or other application services. They can shorten software release cycles and work on updates quickly with the container model. https://aws.amazon.com/what-is/containerization/  Linux containers are made up of control groups (cgroups) and namespaces that help limit what a container can access, but all containers share the same Linux kernel as the host Amazon EC2 instance. https://docs.aws.amazon.com/eks/latest/userguide/security.html

Claim 10	Accused Instrumentalities
	 **Use Windows containers instead of running many applications on one instance of IIS** Consider the following advantages of using Windows containers instead of running multiple applications on one EC2 Windows instance with Internet Information Services (IIS): - **Security** – Containers provide a level of security out of the box that isn't achieved through isolation at the IIS level. If one IIS website or application is compromised, all the other hosted sites are exposed and vulnerable. Container escape is rare and a harder vulnerability to exploit than gaining control of a server through a web vulnerability. - **Flexibility** – The ability to run containers in process isolation and have their own instance allows for more granular networking options. Containers also offer complex distribution methods across many EC2 instances. You don't get these benefits when you consolidate applications on a single IIS instance. - **Management overhead** – Server Name Indication (SNI) creates overhead that requires management and automation. Also, you have to grapple with typical operating system management operations like patching, troubleshooting BSOD (if auto scaling isn't in place), endpoint protection, and so on. Configuring IIS sites according to security best practices is a time consuming and ongoing activity. You might even need to set up trust levels, which also adds to management overhead. Containers are designed to be stateless and immutable. Ultimately, your deployments are faster, more secure, and repeatable if you use Windows containers instead. https://docs.aws.amazon.com/prescriptive-guidance/latest/optimize-costs-microsoft-workloads/windows-containers-main.html

Claim 18

Claim 18	Accused Instrumentalities
18. A computer system as defined in claim 2 wherein SLCSEs are not copies of OSCSEs.	Each Accused Instrumentality comprises or constitutes a computer system as defined in claim 2 wherein SLCSEs are not copies of OSCSEs. For example, in a typical case the SLCSEs come from a Linux distribution independent of the host operating system, and thus are not identical to the OSCSEs. For another example, the SLCSEs are provided to the computer system through a separate process than the process by which the OSCSEs are provided to the computer system, and thus are not copied from the OSCSEs. *See, e.g.*:  Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/  Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. ttps://www.techtarget.com/searchitoperations/definition/Docker-image

Claim 18	Accused Instrumentalities
	 Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image. Thin R/W layer Thin R/W layer Thin R/W layer Thin R/W layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 https://docs.docker.com/storage/storagedriver/